

---

# **simple-dmrg Documentation**

*Release 1.0*

**James R. Garrison and Ryan V. Mishmash**

Nov 03, 2017



---

# Contents

---

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Authors</b>                           | <b>3</b> |
| <b>2</b> | <b>Contents</b>                          | <b>5</b> |
| 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)

Licensed under the MIT license. If you plan to publish work based on this code, please contact us to find out how to cite us.



### 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  $\rho$  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  $\rho$ . 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  $\rho$ . At this point, it’s important to fully understand *why*  $\rho$  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 possible 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 possible 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 Taipei 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)

```

1  #!/usr/bin/env python
2  #
3  # Simple DMRG tutorial. This code contains a basic implementation of the
4  # infinite system algorithm
5  #
6  # Copyright 2013 James R. Garrison and Ryan V. Mishmash.
7  # Open source under the MIT license. Source code at
8  # <https://github.com/simple-dmrg/simple-dmrg/>
9
10 # This code will run under any version of Python >= 2.6. The following line
11 # provides consistency between python2 and python3.
12 from __future__ import print_function, division # requires Python >= 2.6
13
14 # numpy and scipy imports
15 import numpy as np
16 from scipy.sparse import kron, identity
17 from scipy.sparse.linalg import eigsh # Lanczos routine from ARPACK
18
19 # We will use python's "namedtuple" to represent the Block and EnlargedBlock
20 # objects
21 from collections import namedtuple

```

```

22 Block = namedtuple("Block", ["length", "basis_size", "operator_dict"])
23 EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict"])
24
25
26 def is_valid_block(block):
27     for op in block.operator_dict.values():
28         if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
29             return False
30     return True
31
32 # This function should test the same exact things, so there is no need to
33 # repeat its definition.
34 is_valid_enlarged_block = is_valid_block
35
36 # Model-specific code for the Heisenberg XXZ chain
37 model_d = 2 # single-site basis size
38
39 Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d') # single-site S^z
40 Sp1 = np.array([[0, 1], [0, 0]], dtype='d') # single-site S^+
41
42 H1 = np.array([[0, 0], [0, 0]], dtype='d') # single-site portion of H is zero
43
44 def H2(Sz1, Sp1, Sz2, Sp2): # two-site part of H
45     """Given the operators S^z and S^+ on two sites in different Hilbert spaces
46     (e.g. two blocks), returns a Kronecker product representing the
47     corresponding two-site term in the Hamiltonian that joins the two sites.
48     """
49     J = Jz = 1.
50     return (
51         (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
52         Jz * kron(Sz1, Sz2)
53     )
54
55 # conn refers to the connection operator, that is, the operator on the edge of
56 # the block, on the interior of the chain. We need to be able to represent S^z
57 # and S^+ on that site in the current basis in order to grow the chain.
58 initial_block = Block(length=1, basis_size=model_d, operator_dict={
59     "H": H1,
60     "conn_Sz": Sz1,
61     "conn_Sp": Sp1,
62 })
63
64 def enlarge_block(block):
65     """This function enlarges the provided Block by a single site, returning an
66     EnlargedBlock.
67     """
68     mblock = block.basis_size
69     o = block.operator_dict
70
71     # Create the new operators for the enlarged block. Our basis becomes a
72     # Kronecker product of the Block basis and the single-site basis. NOTE:
73     # `kron` uses the tensor product convention making blocks of the second
74     # array scaled by the first. As such, we adopt this convention for
75     # Kronecker products throughout the code.
76     enlarged_operator_dict = {
77         "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),

```

```

78     "conn_Sz": kron(identity(mblock), Sz1),
79     "conn_Sp": kron(identity(mblock), Sp1),
80 }
81
82     return EnlargedBlock(length=(block.length + 1),
83                          basis_size=(block.basis_size * model_d),
84                          operator_dict=enlarged_operator_dict)
85
86 def rotate_and_truncate(operator, transformation_matrix):
87     """Transforms the operator to the new (possibly truncated) basis given by
88     `transformation_matrix`.
89     """
90     return transformation_matrix.conjugate().transpose().dot(operator.
91 ↪dot(transformation_matrix))
92
93 def single_dmrg_step(sys, env, m):
94     """Performs a single DMRG step using `sys` as the system and `env` as the
95     environment, keeping a maximum of `m` states in the new basis.
96     """
97     assert is_valid_block(sys)
98     assert is_valid_block(env)
99
100     # Enlarge each block by a single site.
101     sys_enl = enlarge_block(sys)
102     if sys is env: # no need to recalculate a second time
103         env_enl = sys_enl
104     else:
105         env_enl = enlarge_block(env)
106
107     assert is_valid_enlarged_block(sys_enl)
108     assert is_valid_enlarged_block(env_enl)
109
110     # Construct the full superblock Hamiltonian.
111     m_sys_enl = sys_enl.basis_size
112     m_env_enl = env_enl.basis_size
113     sys_enl_op = sys_enl.operator_dict
114     env_enl_op = env_enl.operator_dict
115     superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) +_
116 ↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
117     H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
118 ↪op["conn_Sz"], env_enl_op["conn_Sp"])
119
120     # Call ARPACK to find the superblock ground state. ("SA" means find the
121     # "smallest in amplitude" eigenvalue.)
122     (energy,), psi0 = eigsh(superblock_hamiltonian, k=1, which="SA")
123
124     # Construct the reduced density matrix of the system by tracing out the
125     # environment
126     #
127     # We want to make the (sys, env) indices correspond to (row, column) of a
128     # matrix, respectively. Since the environment (column) index updates most
129     # quickly in our Kronecker product structure, psi0 is thus row-major ("C
130     # style").
131     psi0 = psi0.reshape([sys_enl.basis_size, -1], order="C")
132     rho = np.dot(psi0, psi0.conjugate().transpose())
133
134     # Diagonalize the reduced density matrix and sort the eigenvectors by
135     # eigenvalue.

```

```

133     evals, evecs = np.linalg.eigh(rho)
134     possible_eigenstates = []
135     for eval, evec in zip(evals, evecs.transpose()):
136         possible_eigenstates.append((eval, evec))
137     possible_eigenstates.sort(reverse=True, key=lambda x: x[0]) # largest eigenvalue_
↪first
138
139     # Build the transformation matrix from the `m` overall most significant
140     # eigenvectors.
141     my_m = min(len(possible_eigenstates), m)
142     transformation_matrix = np.zeros((sys_enl.basis_size, my_m), dtype='d', order='F')
143     for i, (eval, evec) in enumerate(possible_eigenstates[:my_m]):
144         transformation_matrix[:, i] = evec
145
146     truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
147     print("truncation error:", truncation_error)
148
149     # Rotate and truncate each operator.
150     new_operator_dict = {}
151     for name, op in sys_enl.operator_dict.items():
152         new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)
153
154     newblock = Block(length=sys_enl.length,
155                     basis_size=my_m,
156                     operator_dict=new_operator_dict)
157
158     return newblock, energy
159
160 def infinite_system_algorithm(L, m):
161     block = initial_block
162     # Repeatedly enlarge the system by performing a single DMRG step, using a
163     # reflection of the current block as the environment.
164     while 2 * block.length < L:
165         print("L =", block.length * 2 + 2)
166         block, energy = single_dmrg_step(block, block, m=m)
167         print("E/L =", energy / (block.length * 2))
168
169 if __name__ == "__main__":
170     np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)
171
172     infinite_system_algorithm(L=100, m=20)

```

## 2.5.2 simple\_dmrg\_02\_finite\_system.py

(Raw download)

```

1  #!/usr/bin/env python
2  #
3  # Simple DMRG tutorial. This code integrates the following concepts:
4  # - Infinite system algorithm
5  # - Finite system algorithm
6  #
7  # Copyright 2013 James R. Garrison and Ryan V. Mishmash.
8  # Open source under the MIT license. Source code at
9  # <https://github.com/simple-dmrg/simple-dmrg/>
10

```

```

11 # This code will run under any version of Python >= 2.6. The following line
12 # provides consistency between python2 and python3.
13 from __future__ import print_function, division # requires Python >= 2.6
14
15 # numpy and scipy imports
16 import numpy as np
17 from scipy.sparse import kron, identity
18 from scipy.sparse.linalg import eigsh # Lanczos routine from ARPACK
19
20 # We will use python's "namedtuple" to represent the Block and EnlargedBlock
21 # objects
22 from collections import namedtuple
23
24 Block = namedtuple("Block", ["length", "basis_size", "operator_dict"])
25 EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict"])
26
27 def is_valid_block(block):
28     for op in block.operator_dict.values():
29         if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
30             return False
31     return True
32
33 # This function should test the same exact things, so there is no need to
34 # repeat its definition.
35 is_valid_enlarged_block = is_valid_block
36
37 # Model-specific code for the Heisenberg XXZ chain
38 model_d = 2 # single-site basis size
39
40 Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d') # single-site S^z
41 Sp1 = np.array([[0, 1], [0, 0]], dtype='d') # single-site S^+
42
43 H1 = np.array([[0, 0], [0, 0]], dtype='d') # single-site portion of H is zero
44
45 def H2(Sz1, Sp1, Sz2, Sp2): # two-site part of H
46     """Given the operators S^z and S^+ on two sites in different Hilbert spaces
47     (e.g. two blocks), returns a Kronecker product representing the
48     corresponding two-site term in the Hamiltonian that joins the two sites.
49     """
50     J = Jz = 1.
51     return (
52         (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
53     ↪transpose(), Sp2)) +
54         Jz * kron(Sz1, Sz2)
55     )
56
57 # conn refers to the connection operator, that is, the operator on the edge of
58 # the block, on the interior of the chain. We need to be able to represent S^z
59 # and S^+ on that site in the current basis in order to grow the chain.
60 initial_block = Block(length=1, basis_size=model_d, operator_dict={
61     "H": H1,
62     "conn_Sz": Sz1,
63     "conn_Sp": Sp1,
64 })
65
66 def enlarge_block(block):
67     """This function enlarges the provided Block by a single site, returning an
68     EnlargedBlock.

```

```

68     """
69     mblock = block.basis_size
70     o = block.operator_dict
71
72     # Create the new operators for the enlarged block. Our basis becomes a
73     # Kronecker product of the Block basis and the single-site basis. NOTE:
74     # `kron` uses the tensor product convention making blocks of the second
75     # array scaled by the first. As such, we adopt this convention for
76     # Kronecker products throughout the code.
77     enlarged_operator_dict = {
78         "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),
79         "conn_Sz": kron(identity(mblock), Sz1),
80         "conn_Sp": kron(identity(mblock), Sp1),
81     }
82
83     return EnlargedBlock(length=(block.length + 1),
84                          basis_size=(block.basis_size * model_d),
85                          operator_dict=enlarged_operator_dict)
86
87 def rotate_and_truncate(operator, transformation_matrix):
88     """Transforms the operator to the new (possibly truncated) basis given by
89     `transformation_matrix`.
90     """
91     return transformation_matrix.conjugate().transpose().dot(operator.
↪dot(transformation_matrix))
92
93 def single_dmrg_step(sys, env, m):
94     """Performs a single DMRG step using `sys` as the system and `env` as the
95     environment, keeping a maximum of `m` states in the new basis.
96     """
97     assert is_valid_block(sys)
98     assert is_valid_block(env)
99
100    # Enlarge each block by a single site.
101    sys_enl = enlarge_block(sys)
102    if sys is env: # no need to recalculate a second time
103        env_enl = sys_enl
104    else:
105        env_enl = enlarge_block(env)
106
107    assert is_valid_enlarged_block(sys_enl)
108    assert is_valid_enlarged_block(env_enl)
109
110    # Construct the full superblock Hamiltonian.
111    m_sys_enl = sys_enl.basis_size
112    m_env_enl = env_enl.basis_size
113    sys_enl_op = sys_enl.operator_dict
114    env_enl_op = env_enl.operator_dict
115    superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) +_
↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
116        H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
↪op["conn_Sz"], env_enl_op["conn_Sp"])
117
118    # Call ARPACK to find the superblock ground state. ("SA" means find the
119    # "smallest in amplitude" eigenvalue.)
120    (energy,) , psi0 = eigsh(superblock_hamiltonian, k=1, which="SA")
121

```

```

122 # Construct the reduced density matrix of the system by tracing out the
123 # environment
124 #
125 # We want to make the (sys, env) indices correspond to (row, column) of a
126 # matrix, respectively. Since the environment (column) index updates most
127 # quickly in our Kronecker product structure, psi0 is thus row-major ("C
128 # style").
129 psi0 = psi0.reshape([sys_enl.basis_size, -1], order="C")
130 rho = np.dot(psi0, psi0.conjugate().transpose())
131
132 # Diagonalize the reduced density matrix and sort the eigenvectors by
133 # eigenvalue.
134 evals, evecs = np.linalg.eigh(rho)
135 possible_eigenstates = []
136 for eval, evec in zip(evals, evecs.transpose()):
137     possible_eigenstates.append((eval, evec))
138 possible_eigenstates.sort(reverse=True, key=lambda x: x[0]) # largest eigenvalue_
↪first
139
140 # Build the transformation matrix from the `m` overall most significant
141 # eigenvectors.
142 my_m = min(len(possible_eigenstates), m)
143 transformation_matrix = np.zeros((sys_enl.basis_size, my_m), dtype='d', order='F')
144 for i, (eval, evec) in enumerate(possible_eigenstates[:my_m]):
145     transformation_matrix[:, i] = evec
146
147 truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
148 print("truncation error:", truncation_error)
149
150 # Rotate and truncate each operator.
151 new_operator_dict = {}
152 for name, op in sys_enl.operator_dict.items():
153     new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)
154
155 newblock = Block(length=sys_enl.length,
156                 basis_size=my_m,
157                 operator_dict=new_operator_dict)
158
159 return newblock, energy
160
161 def graphic(sys_block, env_block, sys_label="l"):
162     """Returns a graphical representation of the DMRG step we are about to
163     perform, using '=' to represent the system sites, '-' to represent the
164     environment sites, and '**' to represent the two intermediate sites.
165     """
166     assert sys_label in ("l", "r")
167     graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
168     if sys_label == "r":
169         # The system should be on the right and the environment should be on
170         # the left, so reverse the graphic.
171         graphic = graphic[::-1]
172     return graphic
173
174 def infinite_system_algorithm(L, m):
175     block = initial_block
176     # Repeatedly enlarge the system by performing a single DMRG step, using a
177     # reflection of the current block as the environment.
178     while 2 * block.length < L:

```

```

179     print("L =", block.length * 2 + 2)
180     block, energy = single_dmrg_step(block, block, m=m)
181     print("E/L =", energy / (block.length * 2))
182
183 def finite_system_algorithm(L, m_warmup, m_sweep_list):
184     assert L % 2 == 0 # require that L is an even number
185
186     # To keep things simple, this dictionary is not actually saved to disk, but
187     # we use it to represent persistent storage.
188     block_disk = {} # "disk" storage for Block objects
189
190     # Use the infinite system algorithm to build up to desired size. Each time
191     # we construct a block, we save it for future reference as both a left
192     # ("l") and right ("r") block, as the infinite system algorithm assumes the
193     # environment is a mirror image of the system.
194     block = initial_block
195     block_disk["l", block.length] = block
196     block_disk["r", block.length] = block
197     while 2 * block.length < L:
198         # Perform a single DMRG step and save the new Block to "disk"
199         print(graphic(block, block))
200         block, energy = single_dmrg_step(block, block, m=m_warmup)
201         print("E/L =", energy / (block.length * 2))
202         block_disk["l", block.length] = block
203         block_disk["r", block.length] = block
204
205     # Now that the system is built up to its full size, we perform sweeps using
206     # the finite system algorithm. At first the left block will act as the
207     # system, growing at the expense of the right block (the environment), but
208     # once we come to the end of the chain these roles will be reversed.
209     sys_label, env_label = "l", "r"
210     sys_block = block; del block # rename the variable
211     for m in m_sweep_list:
212         while True:
213             # Load the appropriate environment block from "disk"
214             env_block = block_disk[env_label, L - sys_block.length - 2]
215             if env_block.length == 1:
216                 # We've come to the end of the chain, so we reverse course.
217                 sys_block, env_block = env_block, sys_block
218                 sys_label, env_label = env_label, sys_label
219
220             # Perform a single DMRG step.
221             print(graphic(sys_block, env_block, sys_label))
222             sys_block, energy = single_dmrg_step(sys_block, env_block, m=m)
223
224             print("E/L =", energy / L)
225
226             # Save the block from this step to disk.
227             block_disk[sys_label, sys_block.length] = sys_block
228
229             # Check whether we just completed a full sweep.
230             if sys_label == "l" and 2 * sys_block.length == L:
231                 break # escape from the "while True" loop
232
233 if __name__ == "__main__":
234     np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)
235
236     #infinite_system_algorithm(L=100, m=20)

```

```
237 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)

```

1  #!/usr/bin/env python
2  #
3  # Simple DMRG tutorial. This code integrates the following concepts:
4  # - Infinite system algorithm
5  # - Finite system algorithm
6  # - Conserved quantum numbers
7  #
8  # Copyright 2013 James R. Garrison and Ryan V. Mishmash.
9  # Open source under the MIT license. Source code at
10 # <https://github.com/simple-dmrg/simple-dmrg/>
11
12 # This code will run under any version of Python >= 2.6. The following line
13 # provides consistency between python2 and python3.
14 from __future__ import print_function, division # requires Python >= 2.6
15
16 # numpy and scipy imports
17 import numpy as np
18 from scipy.sparse import kron, identity, lil_matrix
19 from scipy.sparse.linalg import eigsh # Lanczos routine from ARPACK
20
21 # We will use python's "namedtuple" to represent the Block and EnlargedBlock
22 # objects
23 from collections import namedtuple
24
25 Block = namedtuple("Block", ["length", "basis_size", "operator_dict", "basis_sector_
↪array"])
26 EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict",
↪"basis_sector_array"])
27
28 def is_valid_block(block):
29     if len(block.basis_sector_array) != block.basis_size:
30         return False
31     for op in block.operator_dict.values():
32         if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
33             return False
34     return True
35
36 # This function should test the same exact things, so there is no need to
37 # repeat its definition.
38 is_valid_enlarged_block = is_valid_block
39
40 # Model-specific code for the Heisenberg XXZ chain
41 model_d = 2 # single-site basis size
42 single_site_sectors = np.array([0.5, -0.5]) # S^z sectors corresponding to the
43 # single site basis elements
44
45 Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d') # single-site S^z
46 Sp1 = np.array([[0, 1], [0, 0]], dtype='d') # single-site S^+
47
48 H1 = np.array([[0, 0], [0, 0]], dtype='d') # single-site portion of H is zero

```

```

49
50 def H2(Sz1, Sp1, Sz2, Sp2): # two-site part of H
51     """Given the operators  $S^z$  and  $S^+$  on two sites in different Hilbert spaces
52     (e.g. two blocks), returns a Kronecker product representing the
53     corresponding two-site term in the Hamiltonian that joins the two sites.
54     """
55     J = Jz = 1.
56     return (
57         (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
58         Jz * kron(Sz1, Sz2)
59     )
60
61 # conn refers to the connection operator, that is, the operator on the edge of
62 # the block, on the interior of the chain. We need to be able to represent  $S^z$ 
63 # and  $S^+$  on that site in the current basis in order to grow the chain.
64 initial_block = Block(length=1, basis_size=model_d, operator_dict={
65     "H": H1,
66     "conn_Sz": Sz1,
67     "conn_Sp": Sp1,
68 }, basis_sector_array=single_site_sectors)
69
70 def enlarge_block(block):
71     """This function enlarges the provided Block by a single site, returning an
72     EnlargedBlock.
73     """
74     mblock = block.basis_size
75     o = block.operator_dict
76
77     # Create the new operators for the enlarged block. Our basis becomes a
78     # Kronecker product of the Block basis and the single-site basis. NOTE:
79     # `kron` uses the tensor product convention making blocks of the second
80     # array scaled by the first. As such, we adopt this convention for
81     # Kronecker products throughout the code.
82     enlarged_operator_dict = {
83         "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),
84         "conn_Sz": kron(identity(mblock), Sz1),
85         "conn_Sp": kron(identity(mblock), Sp1),
86     }
87
88     # This array keeps track of which sector each element of the new basis is
89     # in. `np.add.outer()` creates a matrix that adds each element of the
90     # first vector with each element of the second, which when flattened
91     # contains the sector of each basis element in the above Kronecker product.
92     enlarged_basis_sector_array = np.add.outer(block.basis_sector_array, single_site_
↪sectors).flatten()
93
94     return EnlargedBlock(length=(block.length + 1),
95                          basis_size=(block.basis_size * model_d),
96                          operator_dict=enlarged_operator_dict,
97                          basis_sector_array=enlarged_basis_sector_array)
98
99 def rotate_and_truncate(operator, transformation_matrix):
100     """Transforms the operator to the new (possibly truncated) basis given by
101     `transformation_matrix`.
102     """
103     return transformation_matrix.conjugate().transpose().dot(operator.
↪dot(transformation_matrix))

```

```

104
105 def index_map(array):
106     """Given an array, returns a dictionary that allows quick access to the
107     indices at which a given value occurs.
108
109     Example usage:
110
111     >>> by_index = index_map([3, 5, 5, 7, 3])
112     >>> by_index[3]
113     [0, 4]
114     >>> by_index[5]
115     [1, 2]
116     >>> by_index[7]
117     [3]
118     """
119     d = {}
120     for index, value in enumerate(array):
121         d.setdefault(value, []).append(index)
122     return d
123
124 def single_dmrg_step(sys, env, m, target_Sz):
125     """Performs a single DMRG step using `sys` as the system and `env` as the
126     environment, keeping a maximum of `m` states in the new basis.
127     """
128     assert is_valid_block(sys)
129     assert is_valid_block(env)
130
131     # Enlarge each block by a single site.
132     sys_enl = enlarge_block(sys)
133     sys_enl_basis_by_sector = index_map(sys_enl.basis_sector_array)
134     if sys is env: # no need to recalculate a second time
135         env_enl = sys_enl
136         env_enl_basis_by_sector = sys_enl_basis_by_sector
137     else:
138         env_enl = enlarge_block(env)
139         env_enl_basis_by_sector = index_map(env_enl.basis_sector_array)
140
141     assert is_valid_enlarged_block(sys_enl)
142     assert is_valid_enlarged_block(env_enl)
143
144     # Construct the full superblock Hamiltonian.
145     m_sys_enl = sys_enl.basis_size
146     m_env_enl = env_enl.basis_size
147     sys_enl_op = sys_enl.operator_dict
148     env_enl_op = env_enl.operator_dict
149     superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) + \
150     ↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
151     ↪H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
152     ↪op["conn_Sz"], env_enl_op["conn_Sp"])
153
154     # Build up a "restricted" basis of states in the target sector and
155     # reconstruct the superblock Hamiltonian in that sector.
156     sector_indices = {} # will contain indices of the new (restricted) basis
157     # for which the enlarged system is in a given sector
158     restricted_basis_indices = [] # will contain indices of the old (full) basis,
159     ↪which we are mapping to
160     for sys_enl_Sz, sys_enl_basis_states in sys_enl_basis_by_sector.items():
161         sector_indices[sys_enl_Sz] = []

```

```

159     env_enl_Sz = target_Sz - sys_enl_Sz
160     if env_enl_Sz in env_enl_basis_by_sector:
161         for i in sys_enl_basis_states:
162             i_offset = m_env_enl * i # considers the tensor product structure of
↳the superblock basis
163             for j in env_enl_basis_by_sector[env_enl_Sz]:
164                 current_index = len(restricted_basis_indices) # about-to-be-
↳added index of restricted_basis_indices
165                 sector_indices[sys_enl_Sz].append(current_index)
166                 restricted_basis_indices.append(i_offset + j)
167
168     restricted_superblock_hamiltonian = superblock_hamiltonian[:, restricted_basis_
↳indices][restricted_basis_indices, :]
169
170     # Call ARPACK to find the superblock ground state. ("SA" means find the
171     # "smallest in amplitude" eigenvalue.)
172     (energy,) , restricted_psi0 = eigsh(restricted_superblock_hamiltonian, k=1, which=
↳"SA")
173
174     # Construct each block of the reduced density matrix of the system by
175     # tracing out the environment
176     rho_block_dict = {}
177     for sys_enl_Sz, indices in sector_indices.items():
178         if indices: # if indices is nonempty
179             psi0_sector = restricted_psi0[indices, :]
180             # We want to make the (sys, env) indices correspond to (row,
181             # column) of a matrix, respectively. Since the environment
182             # (column) index updates most quickly in our Kronecker product
183             # structure, psi0_sector is thus row-major ("C style").
184             psi0_sector = psi0_sector.reshape([len(sys_enl_basis_by_sector[sys_enl_
↳Sz]), -1], order="C")
185             rho_block_dict[sys_enl_Sz] = np.dot(psi0_sector, psi0_sector.conjugate().
↳transpose())
186
187     # Diagonalize each block of the reduced density matrix and sort the
188     # eigenvectors by eigenvalue.
189     possible_eigenstates = []
190     for Sz_sector, rho_block in rho_block_dict.items():
191         evals, evecs = np.linalg.eigh(rho_block)
192         current_sector_basis = sys_enl_basis_by_sector[Sz_sector]
193         for eval, evec in zip(evals, evecs.transpose()):
194             possible_eigenstates.append((eval, evec, Sz_sector, current_sector_basis))
195     possible_eigenstates.sort(reverse=True, key=lambda x: x[0]) # largest eigenvalue
↳first
196
197     # Build the transformation matrix from the `m` overall most significant
198     # eigenvectors. It will have sparse structure due to the conserved quantum
199     # number.
200     my_m = min(len(possible_eigenstates), m)
201     transformation_matrix = lil_matrix((sys_enl.basis_size, my_m), dtype='d')
202     new_sector_array = np.zeros((my_m,), dtype='d') # lists the sector of each
203     # element of the new/truncated
↳basis
204     for i, (eval, evec, Sz_sector, current_sector_basis) in enumerate(possible_
↳eigenstates[:my_m]):
205         for j, v in zip(current_sector_basis, evec):
206             transformation_matrix[j, i] = v
207         new_sector_array[i] = Sz_sector

```

```

208 # Convert the transformation matrix to a more efficient internal
209 # representation. `lil_matrix` is good for constructing a sparse matrix
210 # efficiently, but `csr_matrix` is better for performing quick
211 # multiplications.
212 transformation_matrix = transformation_matrix.tocsr()
213
214 truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
215 print("truncation error:", truncation_error)
216
217 # Rotate and truncate each operator.
218 new_operator_dict = {}
219 for name, op in sys_enl.operator_dict.items():
220     new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)
221
222 newblock = Block(length=sys_enl.length,
223                 basis_size=my_m,
224                 operator_dict=new_operator_dict,
225                 basis_sector_array=new_sector_array)
226
227 return newblock, energy
228
229 def graphic(sys_block, env_block, sys_label="l"):
230     """Returns a graphical representation of the DMRG step we are about to
231     perform, using '=' to represent the system sites, '-' to represent the
232     environment sites, and '**' to represent the two intermediate sites.
233     """
234     assert sys_label in ("l", "r")
235     graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
236     if sys_label == "r":
237         # The system should be on the right and the environment should be on
238         # the left, so reverse the graphic.
239         graphic = graphic[::-1]
240     return graphic
241
242 def infinite_system_algorithm(L, m, target_Sz):
243     block = initial_block
244     # Repeatedly enlarge the system by performing a single DMRG step, using a
245     # reflection of the current block as the environment.
246     while 2 * block.length < L:
247         current_L = 2 * block.length + 2 # current superblock length
248         current_target_Sz = int(target_Sz) * current_L // L
249         print("L =", current_L)
250         block, energy = single_dmrg_step(block, block, m=m, target_Sz=current_target_
↪Sz)
251         print("E/L =", energy / current_L)
252
253 def finite_system_algorithm(L, m_warmup, m_sweep_list, target_Sz):
254     assert L % 2 == 0 # require that L is an even number
255
256     # To keep things simple, this dictionary is not actually saved to disk, but
257     # we use it to represent persistent storage.
258     block_disk = {} # "disk" storage for Block objects
259
260     # Use the infinite system algorithm to build up to desired size. Each time
261     # we construct a block, we save it for future reference as both a left
262     # ("l") and right ("r") block, as the infinite system algorithm assumes the
263     # environment is a mirror image of the system.
264     block = initial_block

```

```

265 block_disk["l", block.length] = block
266 block_disk["r", block.length] = block
267 while 2 * block.length < L:
268     # Perform a single DMRG step and save the new Block to "disk"
269     print(graphic(block, block))
270     current_L = 2 * block.length + 2 # current superblock length
271     current_target_Sz = int(target_Sz) * current_L // L
272     block, energy = single_dmrg_step(block, block, m=m_warmup, target_Sz=current_
↪target_Sz)
273     print("E/L =", energy / current_L)
274     block_disk["l", block.length] = block
275     block_disk["r", block.length] = block
276
277     # Now that the system is built up to its full size, we perform sweeps using
278     # the finite system algorithm. At first the left block will act as the
279     # system, growing at the expense of the right block (the environment), but
280     # once we come to the end of the chain these roles will be reversed.
281     sys_label, env_label = "l", "r"
282     sys_block = block; del block # rename the variable
283     for m in m_sweep_list:
284         while True:
285             # Load the appropriate environment block from "disk"
286             env_block = block_disk[env_label, L - sys_block.length - 2]
287             if env_block.length == 1:
288                 # We've come to the end of the chain, so we reverse course.
289                 sys_block, env_block = env_block, sys_block
290                 sys_label, env_label = env_label, sys_label
291
292             # Perform a single DMRG step.
293             print(graphic(sys_block, env_block, sys_label))
294             sys_block, energy = single_dmrg_step(sys_block, env_block, m=m, target_
↪Sz=target_Sz)
295
296             print("E/L =", energy / L)
297
298             # Save the block from this step to disk.
299             block_disk[sys_label, sys_block.length] = sys_block
300
301             # Check whether we just completed a full sweep.
302             if sys_label == "l" and 2 * sys_block.length == L:
303                 break # escape from the "while True" loop
304
305 if __name__ == "__main__":
306     np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)
307
308     #infinite_system_algorithm(L=100, m=20, target_Sz=0)
309     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)

```

1 #!/usr/bin/env python
2 #
3 # Simple DMRG tutorial. This code integrates the following concepts:

```

```

4 # - Infinite system algorithm
5 # - Finite system algorithm
6 # - Conserved quantum numbers
7 # - Eigenstate prediction
8 #
9 # Copyright 2013 James R. Garrison and Ryan V. Mishmash.
10 # Open source under the MIT license. Source code at
11 # <https://github.com/simple-dmrg/simple-dmrg/>
12
13 # This code will run under any version of Python >= 2.6. The following line
14 # provides consistency between python2 and python3.
15 from __future__ import print_function, division # requires Python >= 2.6
16
17 # numpy and scipy imports
18 import numpy as np
19 from scipy.sparse import kron, identity, lil_matrix
20 from scipy.sparse.linalg import eigsh # Lanczos routine from ARPACK
21
22 # We will use python's "namedtuple" to represent the Block and EnlargedBlock
23 # objects
24 from collections import namedtuple
25
26 Block = namedtuple("Block", ["length", "basis_size", "operator_dict", "basis_sector_
↳array"])
27 EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict",
↳"basis_sector_array"])
28
29 def is_valid_block(block):
30     if len(block.basis_sector_array) != block.basis_size:
31         return False
32     for op in block.operator_dict.values():
33         if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
34             return False
35     return True
36
37 # This function should test the same exact things, so there is no need to
38 # repeat its definition.
39 is_valid_enlarged_block = is_valid_block
40
41 # Model-specific code for the Heisenberg XXZ chain
42 model_d = 2 # single-site basis size
43 single_site_sectors = np.array([0.5, -0.5]) # S^z sectors corresponding to the
44 # single site basis elements
45
46 Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d') # single-site S^z
47 Sp1 = np.array([[0, 1], [0, 0]], dtype='d') # single-site S^+
48
49 H1 = np.array([[0, 0], [0, 0]], dtype='d') # single-site portion of H is zero
50
51 def H2(Sz1, Sp1, Sz2, Sp2): # two-site part of H
52     """Given the operators S^z and S^+ on two sites in different Hilbert spaces
53     (e.g. two blocks), returns a Kronecker product representing the
54     corresponding two-site term in the Hamiltonian that joins the two sites.
55     """
56     J = Jz = 1.
57     return (
58         (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↳transpose(), Sp2)) +

```

```

59     Jz * kron(Sz1, Sz2)
60     )
61
62     # conn refers to the connection operator, that is, the operator on the edge of
63     # the block, on the interior of the chain. We need to be able to represent  $S^z$ 
64     # and  $S^+$  on that site in the current basis in order to grow the chain.
65     initial_block = Block(length=1, basis_size=model_d, operator_dict={
66         "H": H1,
67         "conn_Sz": Sz1,
68         "conn_Sp": Sp1,
69     }, basis_sector_array=single_site_sectors)
70
71     def enlarge_block(block):
72         """This function enlarges the provided Block by a single site, returning an
73         EnlargedBlock.
74         """
75         mblock = block.basis_size
76         o = block.operator_dict
77
78         # Create the new operators for the enlarged block. Our basis becomes a
79         # Kronecker product of the Block basis and the single-site basis. NOTE:
80         # `kron` uses the tensor product convention making blocks of the second
81         # array scaled by the first. As such, we adopt this convention for
82         # Kronecker products throughout the code.
83         enlarged_operator_dict = {
84             "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
85     ↪ "conn_Sz"], o["conn_Sp"], Sz1, Sp1),
86             "conn_Sz": kron(identity(mblock), Sz1),
87             "conn_Sp": kron(identity(mblock), Sp1),
88         }
89
90         # This array keeps track of which sector each element of the new basis is
91         # in. `np.add.outer()` creates a matrix that adds each element of the
92         # first vector with each element of the second, which when flattened
93         # contains the sector of each basis element in the above Kronecker product.
94         enlarged_basis_sector_array = np.add.outer(block.basis_sector_array, single_site_
95     ↪ sectors).flatten()
96
97         return EnlargedBlock(length=(block.length + 1),
98                             basis_size=(block.basis_size * model_d),
99                             operator_dict=enlarged_operator_dict,
100                             basis_sector_array=enlarged_basis_sector_array)
101
102     def rotate_and_truncate(operator, transformation_matrix):
103         """Transforms the operator to the new (possibly truncated) basis given by
104         `transformation_matrix`.
105         """
106         return transformation_matrix.conjugate().transpose().dot(operator.
107     ↪ dot(transformation_matrix))
108
109     def index_map(array):
110         """Given an array, returns a dictionary that allows quick access to the
111         indices at which a given value occurs.
112
113         Example usage:
114
115         >>> by_index = index_map([3, 5, 5, 7, 3])
116         >>> by_index[3]

```

```

114     [0, 4]
115     >>> by_index[5]
116     [1, 2]
117     >>> by_index[7]
118     [3]
119     """
120     d = {}
121     for index, value in enumerate(array):
122         d.setdefault(value, []).append(index)
123     return d
124
125 def single_dmrg_step(sys, env, m, target_Sz, psi0_guess=None):
126     """Performs a single DMRG step using `sys` as the system and `env` as the
127     environment, keeping a maximum of `m` states in the new basis. If
128     `psi0_guess` is provided, it will be used as a starting vector for the
129     Lanczos algorithm.
130     """
131     assert is_valid_block(sys)
132     assert is_valid_block(env)
133
134     # Enlarge each block by a single site.
135     sys_enl = enlarge_block(sys)
136     sys_enl_basis_by_sector = index_map(sys_enl.basis_sector_array)
137     if sys is env: # no need to recalculate a second time
138         env_enl = sys_enl
139         env_enl_basis_by_sector = sys_enl_basis_by_sector
140     else:
141         env_enl = enlarge_block(env)
142         env_enl_basis_by_sector = index_map(env_enl.basis_sector_array)
143
144     assert is_valid_enlarged_block(sys_enl)
145     assert is_valid_enlarged_block(env_enl)
146
147     # Construct the full superblock Hamiltonian.
148     m_sys_enl = sys_enl.basis_size
149     m_env_enl = env_enl.basis_size
150     sys_enl_op = sys_enl.operator_dict
151     env_enl_op = env_enl.operator_dict
152     superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) +
153     ↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
154     ↪H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
155     ↪op["conn_Sz"], env_enl_op["conn_Sp"])
156
157     # Build up a "restricted" basis of states in the target sector and
158     # reconstruct the superblock Hamiltonian in that sector.
159     sector_indices = {} # will contain indices of the new (restricted) basis
160     # for which the enlarged system is in a given sector
161     restricted_basis_indices = [] # will contain indices of the old (full) basis,
162     ↪which we are mapping to
163     for sys_enl_Sz, sys_enl_basis_states in sys_enl_basis_by_sector.items():
164         sector_indices[sys_enl_Sz] = []
165         env_enl_Sz = target_Sz - sys_enl_Sz
166         if env_enl_Sz in env_enl_basis_by_sector:
167             for i in sys_enl_basis_states:
168                 i_offset = m_env_enl * i # considers the tensor product structure of
169                 ↪the superblock basis
170                 for j in env_enl_basis_by_sector[env_enl_Sz]:
171                     current_index = len(restricted_basis_indices) # about-to-be-
172                     ↪added index of restricted_basis_indices

```

```

168         sector_indices[sys_enl_Sz].append(current_index)
169         restricted_basis_indices.append(i_offset + j)
170
171     restricted_superblock_hamiltonian = superblock_hamiltonian[:, restricted_basis_
↪indices][restricted_basis_indices, :]
172     if psi0_guess is not None:
173         restricted_psi0_guess = psi0_guess[restricted_basis_indices]
174     else:
175         restricted_psi0_guess = None
176
177     # Call ARPACK to find the superblock ground state. ("SA" means find the
178     # "smallest in amplitude" eigenvalue.)
179     (energy,) = eigsh(restricted_superblock_hamiltonian, k=1, which=
↪"SA", v0=restricted_psi0_guess)
180
181     # Construct each block of the reduced density matrix of the system by
182     # tracing out the environment
183     rho_block_dict = {}
184     for sys_enl_Sz, indices in sector_indices.items():
185         if indices: # if indices is nonempty
186             psi0_sector = restricted_psi0[indices, :]
187             # We want to make the (sys, env) indices correspond to (row,
188             # column) of a matrix, respectively. Since the environment
189             # (column) index updates most quickly in our Kronecker product
190             # structure, psi0_sector is thus row-major ("C style").
191             psi0_sector = psi0_sector.reshape([len(sys_enl_basis_by_sector[sys_enl_
↪Sz]), -1], order="C")
192             rho_block_dict[sys_enl_Sz] = np.dot(psi0_sector, psi0_sector.conjugate().
↪transpose())
193
194     # Diagonalize each block of the reduced density matrix and sort the
195     # eigenvectors by eigenvalue.
196     possible_eigenstates = []
197     for Sz_sector, rho_block in rho_block_dict.items():
198         evals, evecs = np.linalg.eigh(rho_block)
199         current_sector_basis = sys_enl_basis_by_sector[Sz_sector]
200         for eval, evec in zip(evals, evecs.transpose()):
201             possible_eigenstates.append((eval, evec, Sz_sector, current_sector_basis))
202     possible_eigenstates.sort(reverse=True, key=lambda x: x[0]) # largest eigenvalue,
↪first
203
204     # Build the transformation matrix from the `m` overall most significant
205     # eigenvectors. It will have sparse structure due to the conserved quantum
206     # number.
207     my_m = min(len(possible_eigenstates), m)
208     transformation_matrix = lil_matrix((sys_enl.basis_size, my_m), dtype='d')
209     new_sector_array = np.zeros((my_m,), dtype='d') # lists the sector of each
210                                                     # element of the new/truncated
↪basis
211     for i, (eval, evec, Sz_sector, current_sector_basis) in enumerate(possible_
↪eigenstates[:my_m]):
212         for j, v in zip(current_sector_basis, evec):
213             transformation_matrix[j, i] = v
214             new_sector_array[i] = Sz_sector
215     # Convert the transformation matrix to a more efficient internal
216     # representation. `lil_matrix` is good for constructing a sparse matrix
217     # efficiently, but `csr_matrix` is better for performing quick
218     # multiplications.

```

```

219 transformation_matrix = transformation_matrix.tocsr()
220
221 truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
222 print("truncation error:", truncation_error)
223
224 # Rotate and truncate each operator.
225 new_operator_dict = {}
226 for name, op in sys_enl.operator_dict.items():
227     new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)
228
229 newblock = Block(length=sys_enl.length,
230                 basis_size=my_m,
231                 operator_dict=new_operator_dict,
232                 basis_sector_array=new_sector_array)
233
234 # Construct psi0 (that is, in the full superblock basis) so we can use it
235 # later for eigenstate prediction.
236 psi0 = np.zeros([m_sys_enl * m_env_enl, 1], dtype='d')
237 for i, z in enumerate(restricted_basis_indices):
238     psi0[z, 0] = restricted_psi0[i, 0]
239 if psi0_guess is not None:
240     overlap = np.absolute(np.dot(psi0_guess.conjugate().transpose(), psi0).item())
241     overlap /= np.linalg.norm(psi0_guess) * np.linalg.norm(psi0) # normalize it
242     print("overlap |<psi0_guess|psi0>| =", overlap)
243
244 return newblock, energy, transformation_matrix, psi0
245
246 def graphic(sys_block, env_block, sys_label="l"):
247     """Returns a graphical representation of the DMRG step we are about to
248     perform, using '=' to represent the system sites, '-' to represent the
249     environment sites, and '**' to represent the two intermediate sites.
250     """
251     assert sys_label in ("l", "r")
252     graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
253     if sys_label == "r":
254         # The system should be on the right and the environment should be on
255         # the left, so reverse the graphic.
256         graphic = graphic[::-1]
257     return graphic
258
259 def infinite_system_algorithm(L, m, target_Sz):
260     block = initial_block
261     # Repeatedly enlarge the system by performing a single DMRG step, using a
262     # reflection of the current block as the environment.
263     while 2 * block.length < L:
264         current_L = 2 * block.length + 2 # current superblock length
265         current_target_Sz = int(target_Sz) * current_L // L
266         print("L =", current_L)
267         block, energy, transformation_matrix, psi0 = single_dmrg_step(block, block,
↪m=m, target_Sz=current_target_Sz)
268         print("E/L =", energy / current_L)
269
270 def finite_system_algorithm(L, m_warmup, m_sweep_list, target_Sz):
271     assert L % 2 == 0 # require that L is an even number
272
273     # To keep things simple, these dictionaries are not actually saved to disk,
274     # but they are used to represent persistent storage.
275     block_disk = {} # "disk" storage for Block objects

```

```

276 trmat_disk = {} # "disk" storage for transformation matrices
277
278 # Use the infinite system algorithm to build up to desired size. Each time
279 # we construct a block, we save it for future reference as both a left
280 # ("l") and right ("r") block, as the infinite system algorithm assumes the
281 # environment is a mirror image of the system.
282 block = initial_block
283 block_disk["l", block.length] = block
284 block_disk["r", block.length] = block
285 while 2 * block.length < L:
286     # Perform a single DMRG step and save the new Block to "disk"
287     print(graphic(block, block))
288     current_L = 2 * block.length + 2 # current superblock length
289     current_target_Sz = int(target_Sz) * current_L // L
290     block, energy, transformation_matrix, psi0 = single_dmrg_step(block, block,
↳m=m_warmup, target_Sz=current_target_Sz)
291     print("E/L =", energy / current_L)
292     block_disk["l", block.length] = block
293     block_disk["r", block.length] = block
294
295 # Now that the system is built up to its full size, we perform sweeps using
296 # the finite system algorithm. At first the left block will act as the
297 # system, growing at the expense of the right block (the environment), but
298 # once we come to the end of the chain these roles will be reversed.
299 sys_label, env_label = "l", "r"
300 sys_block = block; del block # rename the variable
301 sys_trmat = None
302 for m in m_sweep_list:
303     while True:
304         # Load the appropriate environment block from "disk"
305         env_block = block_disk[env_label, L - sys_block.length - 2]
306         env_trmat = trmat_disk.get((env_label, L - sys_block.length - 1))
307
308         # If possible, predict an estimate of the ground state wavefunction
309         # from the previous step's psi0 and known transformation matrices.
310         if psi0 is None or sys_trmat is None or env_trmat is None:
311             psi0_guess = None
312         else:
313             # psi0 currently looks e.g. like ==**--- but we need to
314             # transform it to look like ==**--- using the relevant
315             # transformation matrices and paying careful attention to the
316             # tensor product structure.
317             #
318             # Keep in mind that the tensor product of the superblock is
319             # (sys_enl_block, env_enl_block), which is equal to
320             # (sys_block, sys_extra_site, env_block, env_extra_site).
321             # Note that this does *not* correspond to left-to-right order
322             # on the chain.
323             #
324             # First we reshape the psi0 vector into a matrix with rows
325             # corresponding to the enlarged system basis and columns
326             # corresponding to the enlarged environment basis.
327             psi0_a = psi0.reshape((-1, env_trmat.shape[1] * model_d), order="C")
328             # Now we transform the enlarged system block into a system
329             # block, so that psi0_b looks like ==**--- (with only one
330             # intermediate site).
331             psi0_b = sys_trmat.conjugate().transpose().dot(psi0_a)
332             # At the moment, the tensor product goes as (sys_block,

```

```

333         # env_enl_block) == (sys_block, env_block, extra_site), but we
334         # need it to look like (sys_enl_block, env_block) ==
335         # (sys_block, extra_site, env_block). In other words, the
336         # single intermediate site should now be part of a new enlarged
337         # system, not part of the enlarged environment.
338         psi0_c = psi0_b.reshape((-1, env_trmat.shape[1], model_d), order="C").
↪transpose(0, 2, 1)
339         # Now we reshape the psi0 vector into a matrix with rows
340         # corresponding to the enlarged system and columns
341         # corresponding to the environment block.
342         psi0_d = psi0_c.reshape((-1, env_trmat.shape[1]), order="C")
343         # Finally, we transform the environment block into the basis of
344         # an enlarged block the so that psi0_guess has the tensor
345         # product structure of =====.
346         psi0_guess = env_trmat.dot(psi0_d.transpose()).transpose().reshape((-
↪1, 1))
347
348         if env_block.length == 1:
349             # We've come to the end of the chain, so we reverse course.
350             sys_block, env_block = env_block, sys_block
351             sys_label, env_label = env_label, sys_label
352             if psi0_guess is not None:
353                 # Re-order psi0_guess based on the new sys, env labels.
354                 psi0_guess = psi0_guess.reshape((sys_trmat.shape[1] * model_d,
↪env_trmat.shape[0]), order="C").transpose().reshape((-1, 1))
355
356             # Perform a single DMRG step.
357             print(graphic(sys_block, env_block, sys_label))
358             sys_block, energy, sys_trmat, psi0 = single_dmrg_step(sys_block, env_
↪block, m=m, target_Sz=target_Sz, psi0_guess=psi0_guess)
359
360             print("E/L =", energy / L)
361
362             # Save the block and transformation matrix from this step to disk.
363             block_disk[sys_label, sys_block.length] = sys_block
364             trmat_disk[sys_label, sys_block.length] = sys_trmat
365
366             # Check whether we just completed a full sweep.
367             if sys_label == "1" and 2 * sys_block.length == L:
368                 break # escape from the "while True" loop
369
370 if __name__ == "__main__":
371     np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)
372
373     #infinite_system_algorithm(L=100, m=20, target_Sz=0)
374     finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40],
↪target_Sz=0)

```