DMRG101 tutorial Documentation

Release 0.1

Ivan Gonzalez

October 29, 2016

Contents

These pages contain a few exercises to be used with the dmrg101 code. The exercises are the ones used for the hands-on tutorial of the Taipei Density Matrix Renormalization Group Winter School. We hope that even those that could not attend to the School can find this useful to learn some basics about the Density Matrix Renormalization Group algorithm (DMRG).

Some of the exercises here are adapted from a similar tutorial by Adrian Feiguin, published in Lectures on the Physics of Strongly Correlated Systems XV.

This page is only about the tutorial itself. If you need information about the drmg101 code, check out the dmrg101 documentation.

There is also a PDF version of this webpage.

CHAPTER 1

Warmup

1.1 How to use the tutorial

Each of the exercises is more or less self-content, but ideally you will do them in order as the last exercises require more familiarity with the code.

You are supposed to write some code on your own and then run it to finish the exercises. Obviously you don't have to write the whole code on your own, you are supposed to use the functions that we already coded within the dmrg101 package. As each of you have a different set of programming skills, the amount of guidance you need to complete the exercises will be different in each case.

We provide a complete implementation of the code needed for each exercise, so nobody gets stuck in the programming part. Just find your pace and use the solution and hints as much as you need it. Remember that this is not a course on programming, but on physics, so focus in understanding the algorithm and the physical ideas behind it.

If you installed dmrg101, you have a copy of all the solution for the exercises. Just check now that everything is there:

```
$ cd tutorial
$ ls
```

We recommmend that you don't overwrite these files, so you have always a code for the exercise that works. If you want to use the solution as an starting point for you own coding, just make a new folder and copy the solution in there.

Let's start!

1.2 Playing with a two-qbit system

The goal of this exercise is to build the wavefunction of a pair of spins one-half (a.k.a. a pair of qbits), and calculate its entanglement entropy. This is a pretty trivial exercise that you could do without much hassle in a piece of paper (see below.) The purpose is just get you familiar with how things are done in the code before moving to bigger adventures.

1.2.1 Exercise

Calculate the entanglement entropy when you trace out one of the spins in a general state in the subspace of a two spin one-half system where the two particles have opposite spin. For which of the states in this subspace is are the two spins maximally entangled (i.e. the Von Neumann entanglement entropy is maximal)?

1.2.2 Hint

The first thing we need to do is to write the wavefunction of the two spin system. In dmrg101 the wavefunctions are represented as matrices instead of vectors, which may be more familiar to you.

The reason for that is that as in DMRG we always have to split the physical systems (say a chain of spins, or the two spins of the problem) in left and right subsystems, the notation with matrices is more suited. In the dmrg101 code the rows of the matrix representing a wavefunction correspond to states of the left subsystem, and the columns correspond to states of the right subsystem.

For example to represent the two spin one-half system, with one spin as left subsystem and the other as right subsystem, we need a 2x2 matrix. Matrix elements in the first row (column) will correspond to states with the left (right) spin down. Matrix elements in the second row (column) will correspond to states with the left (right) spin up. The choice of whether the first or second row corresponds to spin down or up is arbitrary, but once you made the choice you have to be consistent.

If we restrict ourselves to the subspace where the particles have opposite spin, the most general wavefunction for the two qbit systems is simply [1]:

$$|\psi\rangle = \cos\phi|\downarrow\uparrow\rangle + \sin\phi|\uparrow\downarrow\rangle = \begin{pmatrix} 0 & \cos\phi \\ \sin\phi & 0 \end{pmatrix}$$

1.2.3 Solution

The plan is the following. First we are going to write a function to calculate the wavefunction for the two-qbit system as a function of an angle *psi*:

```
def create_two_qbit_system_in_singlet(psi):
    """ Returns the wf of the system as a function of `psi`.

    The (normalized) wavefunction of the two-qbit system can be
    parametrized as a function an angle `psi`.

    Parameters
    ----------
    psi : a double
        Parametrizes the wavefunction.

    Returns
    -------
    result : a Wavefunction
        The wavefunction of the two-qbit system for the given `psi`.
    """
    result = Wavefunction(2, 2)

    # set the different components.
    result.as_matrix[0, 0] = 0.
    result.as_matrix[0, 1] = cos(psi)
    result.as_matrix[1, 0] = sin(psi)
    result.as_matrix[1, 1] = 0.
    return result
```

Now we are going to get the reduced density matrix tracing out the left qbit and calculate the corresponding entanglement entropy:

[1] There could be an extra phase between the two components, but it cancels out later, so we don't bother to include it.

```
def trace_out_left_qbit_and_calculate_entropy(wf):
    """Calculates the entropy after tracing out the left qbit.

    To calculate the entanglement entropy you need to first build the
    reduced density matrix tracing out the degrees of freedom of one of
    the two qbits (it does not matter which, we pick up left here.)

    Parameters
    ----------
    wf : a Wavefunction
        The wavefunction you build up the reduced density matrix with.

    Returns
    -------
    result : a double
        The value of the von Neumann entanglement entropy after tracing
        out the left qbit.
    """
    reduced_DM_for_right_qbit = wf.build_reduced_density_matrix('left')
    evals, evecs = diagonalize(reduced_DM_for_right_qbit)
    result = calculate_entropy(evals)
    return result
```

Now it just a matter to generate a bunch of different values for *psi*, calculate the corresponding wavefunction with the first function above, and pass the wavefunction to the second funciton above to get the value for the entropy. The following code makes this:

```
def main(args):
    """Calculates the entanglement entropy for a system of two qbits in a
    singlet state.
    """
    #
    # get a bunch of values (number_of_psi) for psi
    #
    number_of_psi = 1000
    step = 2*pi/number_of_psi
    psi_values = [x*step for x in range(number_of_psi)]
    #
    # python function map applies a function to a sequence
    #
    wfs = map(create_two_qbit_system_in_singlet, psi_values)
    entropies = map(trace_out_left_qbit_and_calculate_entropy, wfs)
    #
    # find to which value of psi corresponds the max entropy
    #
    zipped = zip(psi_values, entropies)
    max_value = max(zipped, key=lambda item: (item[1]))
    #
    # print the results
    #
    print "The maximum value for entropy is %8.6f." %max_value[1]
    print "The wavefunction with max entropy is: "
    print create_two_qbit_system_in_singlet(max_value[0]).as_matrix
    #
    # save for plotting
    #
    filename = args['--output']
    f = open(filename, 'w')
```

```
    f.write('\n'.join('%s %s' % x for x in zipped))
    f.close()
```

See a full implementation of the above code. If you run that code you should get something like this:

```
$ ./solutions/two_qbit_system.py
The maximum value for entropy is 0.693147.
The wavefunction with max entropy is:
[[ 0.          0.70710678]
 [ 0.70710678  0.        ]]
The whole list of psi vs entropies is saved in two_qbit_entropies.dat.
```

which are in fact the entropy ($log(2)$) and the wavefunction of the triplet state where the particles have opposite spins. In your own code, you might alternatively have observed a state with the same entropy but where one of the two components is negative; this is the singlet state. See also the data for the entropies vs psi. To see the options avaliable:

```
$ ./solutions/two_qbit_system.py --help
Calculates the entanglement entropy of a two qbit system

Calculates the von Neumann entanglement entropu of a system of two
spin one-half spins restricted to the subspace of total spin equal to
zero.

Usage:
  two_qbit_system.py [--dir=DIR -o=FILE]
  two_qbit_system.py -h | --help

  Options:
    -h --help         Shows this screen.
    -o --output=FILE  Ouput file [default: two_qbit_entropies.dat]
    --dir=DIR         Ouput directory [default: ./]
```

1.2.4 Conclusion

It is important that you note that this is the general solution for a system of two qbits, and that two-qbits cannot be more entangled than when they are in an equal amplitude (but possibly opposite phase) superposition state over configurations where the particles have opposite spin. In system of many particles is splitted in two parts (think in a larger chain of spins cut at some point in two), one can always represent the relevant degrees of freedom at the cut as a set of qbits. Then it follows from the result you just proved that the most *economical* way of representing the entanglement across the cut is to map the degrees of freedom of each side to a qbits and *maximally entangle* them across the cut. Any other state to be formed with the qbits in one side and the other, will either have less entanglement across the cut than the one in the original degrees of freedom, or use more qbits at each side of the cut. This is the basis of the mappings used in quantum information methods like MPS or TNS, and you will see *maximally entangled spins/qbits* a lot in the rest of the school.

1.3 Heisenberg model for two spins

The goal of this exercise is to build a Hamiltonian for the two spin system in the previous exercise, and calculate its ground state. Again the purpose is just to get you familiar with the code.

The Hamiltonian that we will build is the antiferromagnetic Heisenberg model, which reads like:

$$H = \vec{S}_1 \cdot \vec{S}_2 = S_1^z S_2^z + \frac{1}{2}\left(S_1^\dagger S_2^- + S_1^- S_2^\dagger\right)$$

where $\vec{S}_i = \vec{\sigma}_i/2$ are the spin operators, σ_i are the Pauli matrices, and $S_i^\pm = S_i^x \pm iS_i^y$.

1.3.1 Exercise

Calculate the ground state (energy and wavefunction) of the antiferromagentic Heisenberg model for a system of two spins one-half.

1.3.2 Hint

Two things. First, remember that the wavefunction in the code is written as a matrix, where the left (row) indexes correspond to the left system, i.e. the left spin, and where the right (colum) indexes correspond to the right system, i.e. the right spin.

Second, once we have build up the Hamiltonian we will use the Lanczos algorithm to get its ground state. This is a whole beast on his own, and we are not going to enter much into it. Just believe that it works as described. [1]

1.3.3 Solution

The first thing we need is the spin operators. For this you can create a *SpinOneHalfSite* which is an object in dmrg101 with the operators you need build in:

```
>>> from dmrg101.core.Sites import SpinOneHalfSite
>>> left_spin = SpinOneHalfSite()
>>> right_spin = SpinOneHalfSite()
>>> # check all it's what you expected
>>> print left_spin.operators['s_z']
[[-0.5  0. ]
 [ 0.   0.5]]
>>> print left_spin.operators['s_p']
[[ 0.  0.]
 [ 1.  0.]]
>>> print left_spin.operators['s_m']
[[ 0.  1.]
 [ 0.  0.]]
```

Now we have to build the Hamiltonian operator, keeping in mind the wavefunction is a matrix. There is an *operator* object in the code which takes care of this issue. Basically you create a new operator by creating first a blank operator and adding to it terms. To add a term you have to specify which is the operator acting on the left system, and which on the right system. The following function does that:

```
def build_HAF_hamiltonian_for_two_spins(left_spin, right_spin):
    """ Builds the AF Heisenberg Hamiltonian for two spins.

    Parameters
    ----------
        left_spin : a Site
            The Site must have the s_z, s_p, and s_m operators defined.
        right_spin : a Site
```

[1] The two spin system is small enough to be solve by exact diagonalization, i.e. just diagonalizing fully the Hamiltonian matrix. We use Lanczos here, because the larger systems that we will find later cannot be fully diagonalized, and you're force to stick with Lanczos or a similar method.

```
            The Site must have the s_z, s_p, and s_m operators defined.

    Returns
    -------
        result : an Operator
            The Hamiltonian of the AF Heisenberg.

    Notes
    -----
        This function should raise an exception if the keys for the
        operators are not found in the site, but I'll leave without it
        because it just makes the code more complicated to read.
    """
    result = CompositeOperator(left_spin.dim, right_spin.dim)
    result.add(left_spin.operators['s_z'], right_spin.operators['s_z'])
    result.add(left_spin.operators['s_p'], right_spin.operators['s_m'], .5)
    result.add(left_spin.operators['s_m'], right_spin.operators['s_p'], .5)
    return result
```

Then it's just a matter to call the Lanczos subroutine to solve for the ground state and put everything together:

```
def main():
    #
    # create the two spin one-half sites
    #
    left_spin = SpinOneHalfSite()
    right_spin = SpinOneHalfSite()
    #
    # build the Hamiltonian, and solve it using Lanczos.
    #
    hamiltonian = build_HAF_hamiltonian_for_two_spins(left_spin,
                                                      right_spin)
    ground_state_energy, ground_state_wf = calculate_ground_state(hamiltonian)
    #
    # print results
    #
    print "The ground state energy is %8.6f." %ground_state_energy
    print "The ground state wavefunction is :"
    print ground_state_wf.as_matrix
```

See `a full implementation of the above code`. If you run that code you should get something like this:

```
$ ./solutions/heisenberg_for_two_spins.py
The ground state energy is -0.75
The ground state wavefunction is:
[[ 0.          0.70710678]
 [ 0.70710678  0.        ]]
```

1.4 Heisenberg model for four spins

The goal of this exercise is to build a Hamiltonian for the four spin system, and calculate its ground state. The difference with the previous exercise is that we will use the same setup as for the DMRG algorithm. This means that we will build a chain as a block-site-site-block system, where the left-most spin will be the left block, the two central spins will be the single sites, and the right-most spin will be the right block.

Again, the Hamiltonian that we will build is the antiferromagnetic Heisenberg model:

$$H = \sum_i \vec{S}_i \cdot \vec{S}_{i+1} = \sum_i \left[S_i^z S_{i+1}^z + \frac{1}{2} \left(S_i^\dagger S_{i+1}^- + S_i^- S_{i+1}^\dagger \right) \right]$$

where $\vec{S}_i = \vec{\sigma}_i/2$ are the spin operators, σ_i are the Pauli matrices, and $S_i^\pm = S_i^x \pm i S_i^y$.

1.4.1 Exercise

Calculate the ground state (energy and wavefunction) of the antiferromagnetic Heisenberg model for a system of four spins one-half.

1.4.2 Solution

You can use the *System* class to create the four spin chain, and set the system Hamiltonian to the antiferromagnetic Heisenberg model. To do that you add the terms to the system Hamiltonian, and to add a term you just have to specify which operator acts in each of the parts: left block, left site, right site, and right block, using the name of the operator. You can specify a parameter if you need to with a fifth optional argument. This function does that:

```
def set_hamiltonian_to_AF_Heisenberg(system):
    """Sets a system Hamiltonian to the AF Heisenberg Hamiltonian.

    Does exactly this. If the system hamiltonian has some other terms on
    it, there are not touched. So be sure to use this function only in
    newly created `System` objects.

    Parameters
    ----------
    system : a System.
        The System you want to set the Hamiltonain for.
    """
    system.add_to_hamiltonian('id', 'id', 's_z', 's_z')
    system.add_to_hamiltonian('id', 'id', 's_p', 's_m', .5)
    system.add_to_hamiltonian('id', 'id', 's_m', 's_p', .5)
    system.add_to_hamiltonian('id', 's_z', 's_z', 'id')
    system.add_to_hamiltonian('id', 's_p', 's_m', 'id', .5)
    system.add_to_hamiltonian('id', 's_m', 's_p', 'id', .5)
    system.add_to_hamiltonian('s_z', 's_z', 'id', 'id')
    system.add_to_hamiltonian('s_p', 's_m', 'id', 'id', .5)
    system.add_to_hamiltonian('s_m', 's_p', 'id', 'id', .5)
```

The remaining is calling again to the Lanczos solver to get the ground state energy and wavefunction. You can use a convenience function in the *System* class. Putting everything together you end up with something like this:

```
def main():
    #
    # create a system object with spin one-half sites and blocks.
    #
    spin_one_half_site = SpinOneHalfSite()
    system = System(spin_one_half_site)
    #
    # build the Hamiltonian, and solve it using Lanczos.
    #
    set_hamiltonian_to_AF_Heisenberg(system)
    ground_state_energy, ground_state_wf = system.calculate_ground_state()
    #
```

```
    # print results
    #
    print "The ground state energy is %8.6f." %ground_state_energy
    print "The ground state wavefunction is :"
    print ground_state_wf.as_matrix
```

See `a full implementation of the above code`. If you run that code you should get:

```
$ ./solutions/heisenberg_for_four_spins.py
The ground state energy is -1.616025.
The ground state wavefunction is:
[[ ...
```

CHAPTER 2

Spin systems

2.1 Infinite DMRG algorithm for the Heisenberg chain

The goal of this exercise is to implement the infinite version of the DMRG algorithm for the antiferromagnetic Heisenberg chain of spins one-half.

The Hamiltonian that we will build is the antiferromagnetic Heisenberg model:

$$H = \sum_i \vec{S}_i \cdot \vec{S}_{i+1} = \sum_i \left[S_i^z S_{i+1}^z + \frac{1}{2} \left(S_i^\dagger S_{i+1}^- + S_i^- S_{i+1}^\dagger \right) \right]$$

where $\vec{S}_i = \vec{\sigma}_i/2$ are the spin operators, σ_i are the Pauli matrices, and $S_i^\pm = S_i^x \pm i S_i^y$.

2.1.1 Exercise

Calculate the ground state energy of the antiferromagnetic Heisenberg model for a chain of spins $S = 1/2$ using the infinite version of the DMRG algorithm. You should be able to pass the number of sites of the chain and the number of states kept during the DMRG truncation as parameters. The output should be the energy per site, entanglement entropy and, truncation error at each step of the algorithm. You could:

- make a plot of the energy per site versus system size to see how converges. The relative error in energy is given by the truncation error.
- for a given system size (say $n = 32, \ldots, 64$) calculate the energy per site for different number of states kept, make a plot, and extrapolate the values of the energy to zero truncation error with a linear fit. Compare this with the exact value for a finite size given by the Bethe Anstatz.

L	E/J
16	-6.9117371455749
24	-10.4537857604096
32	-13.9973156182243
48	-21.0859563143863
64	-28.1754248597421

2.1.2 Solution

You can use the *System* class to create the four spin chain, and set the system Hamiltonian to the antiferromagnetic Heisenberg model, as we did in the previous exercise. Then you calculate ground state as before. From there you need to implement the DMRG transformation. We did most of the required steps (calculating, diagonalizing, and truncating the reduced density matrix) in a previous exercise.

The new thing is updating the operators using the transformation matrix. You need to specify two things before doing that. First which are going to be the operators to be transformed. These are the ones that you need in the next step of the DMRG algorithm. To build the Heisenberg Hamiltonian using the block-site-site-block construction, you need your block to have a truncated version of the current single site spin operators $S^z, S^\dagger, S^-$. Additionally you need to update the block hamiltonian, which belongs to the current block and contains the part of the Hamiltonian involving only degrees of freedom *within* the block. This function sets the operators to be updated:

```
def set_operators_to_update_to_AF_Heisenberg(system):
    """Sets the operators to update to be what you need to AF Heisenberg.

    Parameters
    ----------
    system : a System.
        The System you want to set the Hamiltonian for.

    Notes
    -----
    The block Hamiltonian, althought needs to be updated, is treated
    separately by the very functions in the `System` class.
    """
    system.add_to_operators_to_update('s_z', site_op='s_z')
    system.add_to_operators_to_update('s_p', site_op='s_p')
    system.add_to_operators_to_update('s_m', site_op='s_m')
```

The block hamiltonian is built by including the previous block hamiltonian, if any, and the terms coming from the interactions between the block and the single site that is being included in the new block:

```
def set_block_hamiltonian_to_AF_Heisenberg(system):
    """Sets the block Hamiltonian to be what you need for AF Heisenberg.

    Builds a matrix with the proper dimensions full of zeros. Then adds
    the terms in the Hamiltonian that contain degrees of freedom belonging
    only to one block. Finally adds the matrix to the block as the
    operator labelled 'bh'.

    Parameters
    ----------
    system : a System.
        The System you want to set the Hamiltonian for.
    """
    tmp_matrix_size = None
    if system.growing_side == 'left':
        tmp_matrix_size = system.get_left_dim()
    else:
        tmp_matrix_size = system.get_right_dim()
    tmp_matrix_for_bh = np.zeros((tmp_matrix_size, tmp_matrix_size))
    if 'bh' in system.growing_block.operators.keys():
        system.add_to_block_hamiltonian(tmp_matrix_for_bh, 'bh', 'id')
    system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_z', 's_z')
    system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_p', 's_m', .5)
    system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_m', 's_p', .5)
    system.operators_to_add_to_block['bh'] = tmp_matrix_for_bh
```

Now we can put together the function to grow a block by one site:

```
def grow_block_by_one_site(growing_block, ground_state_wf, system,
                           number_of_states_kept):
    """Grows one side of the system by one site.
```

```
    Calculates the truncation matrix by calculating the reduced density
    matrix for `ground_state_wf` by tracing out the degrees of freedom of
    the shrinking side. Then updates the operators you need in the next
    steps, effectively growing the size of the block by one site.

    Parameters
    ----------
    growing_block : a string.
        The block which is growing. It must be 'left' or 'right'.
    ground_state_wf : a Wavefunction.
        The ground state wavefunction of your system.
    system : a System object.
        The system you want to do the calculation on. This function
        assumes that you have set the Hamiltonian to something.
    number_of_states_kept : an int.
        The number of states you want to keep in each block after the
        truncation. If the `number_of_states_kept` is smaller than the
        dimension of the current Hilbert space block, all states are kept.

    Returns
    -------
    entropy : a double.
        The Von Neumann entropy for the cut that splits the chain into two
        equal halves.
    truncation_error : a double.
        The truncation error, i.e. the sum of the discarded eigenvalues of
        the reduced density matrix.

    Raises
    ------
    DMRGException
        if `growing_side` is not 'left' or 'right'.
    """
    if growing_block not in ('left', 'right'):
        raise DMRGException('Growing side must be left or right.')
    system.set_growing_side(growing_block)
    rho = ground_state_wf.build_reduced_density_matrix(growing_block)
    evals, evecs = diagonalize(rho)
    truncated_evals, truncation_matrix = truncate(evals, evecs,
                                                  number_of_states_kept)
    entropy = calculate_entropy(truncated_evals)
    truncation_error = calculate_truncation_error(truncated_evals)
    set_block_hamiltonian_to_AF_Heisenberg(system)
    set_operators_to_update_to_AF_Heisenberg(system)
    system.update_all_operators(truncation_matrix)
    return entropy, truncation_error
```

A step of the infinite DMRG algorithm consists in growing at the same time both the left and the right blocks by one site.

```
def infinite_dmrg_step(system, current_size, number_of_states_kept):
    """Performs one step of the infinite DMRG algorithm.

    Calculates the ground state of a system with a given size, then
    performs the DMRG transformation on the operators of *both* blocks,
    therefore increasing by one site the number of sites encoded in the
    Hilbert space of each blocks, and reset the blocks in the system to be
    the new, enlarged, truncated ones.
```

```
    In reality the second block is not updated but just copied over from
    the first.

    Parameters
    ----------
    system : a System object.
        The system you want to do the calculation on. This function
        assumes that you have set the Hamiltonian to something.
    current_size : an int.
        The number of sites of the chain.
    number_of_states_kept : an int.
        The number of states you want to keep in each block after the
        truncation. If the `number_of_states_kept` is smaller than the
        dimension of the current Hilbert space block, all states are kept.

    Returns
    -------
    energy_per_site : a double.
        The energy per site for the `current_size`.
    entropy : a double.
        The Von Neumann entropy for the cut that splits the chain into two
        equal halves.
    truncation_error : a double.
        The truncation error, i.e. the sum of the discarded eigenvalues of
        the reduced density matrix.

    Notes
    -----
    Normally you don't update both blocks. If the chain is symmetric, you
    just can use the operators for the one of the sides to mirror the
    operators in the other side, saving the half of the CPU time. In
    practical DMRG calculations one uses the finite algorithm to
    improve the result of the infinite algorithm, and one of the blocks
    is kept one site long, and therefore not updated.
    """
    set_hamiltonian_to_AF_Heisenberg(system)
    ground_state_energy, ground_state_wf = system.calculate_ground_state()
    entropy, truncation_error = grow_block_by_one_site('left', ground_state_wf,
                                                       system,
                                                       number_of_states_kept)
    system.right_block = system.left_block
    return ground_state_energy / current_size, entropy, truncation_error
```

The other thing remaining is to make a small change in the function to set up the AF Heisenberg Hamiltonian that we used in the previous exercise to include the block hamiltonians:

```
def set_hamiltonian_to_AF_Heisenberg(system):
    """Sets a system Hamiltonian to the AF Heisenberg Hamiltonian.

    Does exactly this. If the system hamiltonian has some other terms on
    it, there are not touched. So be sure to use this function only in
    newly created `System` objects.

    Parameters
    ----------
    system : a System.
        The System you want to set the Hamiltonain for.
    """
```

```
    system.clear_hamiltonian()
    if 'bh' in system.left_block.operators.keys():
        system.add_to_hamiltonian(left_block_op='bh')
    if 'bh' in system.right_block.operators.keys():
        system.add_to_hamiltonian(right_block_op='bh')
    system.add_to_hamiltonian('id', 'id', 's_z', 's_z')
    system.add_to_hamiltonian('id', 'id', 's_p', 's_m', .5)
    system.add_to_hamiltonian('id', 'id', 's_m', 's_p', .5)
    system.add_to_hamiltonian('id', 's_z', 's_z', 'id')
    system.add_to_hamiltonian('id', 's_p', 's_m', 'id', .5)
    system.add_to_hamiltonian('id', 's_m', 's_p', 'id', .5)
    system.add_to_hamiltonian('s_z', 's_z', 'id', 'id')
    system.add_to_hamiltonian('s_p', 's_m', 'id', 'id', .5)
    system.add_to_hamiltonian('s_m', 's_p', 'id', 'id', .5)
```

Now you just make a loop that repeats the DMRG step until you reach the desired system size. Adding some code to pass the arguments from the command line and to save the results to a file, you get something like this:

```
def main(args):
    #
    # create a system object with spin one-half sites and blocks.
    #
    spin_one_half_site = SpinOneHalfSite()
    system = System(spin_one_half_site)
    #
    # read command-line arguments and initialize some stuff
    #
    number_of_sites = int(args['-n'])
    number_of_states_kept= int(args['-m'])
    sizes = []
    energies = []
    entropies = []
    truncation_errors = []
    #
    # infinite DMRG algorithm
    #
    number_of_sites = 2 * (number_of_sites / 2) # make it even
    for current_size in range(4, number_of_sites + 1, 2):
        #block_size = current_size / 2 - 1
        energy, entropy, truncation_error = (
            infinite_dmrg_step(system, current_size, number_of_states_kept) )
        sizes.append(current_size)
        energies.append(energy)
        entropies.append(entropy)
        truncation_errors.append(truncation_error)
    #
    # save results
    #
    output_file = os.path.join(os.path.abspath(args['--dir']), args['--output'])
    f = open(output_file, 'w')
    zipped = zip (sizes, energies, entropies, truncation_errors)
    f.write('\n'.join('%s %s %s %s' % x for x in zipped))
    f.close()
    print 'Results stored in ' + output_file
```

See `a full implementation of the above code`. To learn how to run this code you can use:

```
$ ./tutorial/solutions/infinite_heisenberg.py --help
Implements the infinite version of the DMRG algorithm for the S=1/2 AF
Heisenberg.

Calculates the ground state energy and wavefunction for the
antiferromagnetic Heisenberg model for a chain of spin one-half. The
calculation of the ground state is done using the infinite version of the
DMRG algorithm.

Usage:
  infinite_heisenberg.py (-m=<states> -n=<sites>) [--dir=DIR -o=FILE]
  infinite_heisenberg.py -h | --help

  Options:
    -h --help         Shows this screen.
    -n <sites>        Number of sites of the chain.
    -m <states>       Number of states kept.
    -o --output=FILE  Ouput file [default: infinite_heisenberg.dat]
    --dir=DIR         Ouput directory [default: ./]
```

2.2 Finite DMRG algorithm for the Heisenberg chain

The goal of this exercise is to implement the finite version of the DMRG algorithm for the antiferromagnetic Heisenberg chain of spins one-half.

The Hamiltonian that we will build is the antiferromagnetic Heisenberg model:

$$H = \sum_i \vec{S}_i \cdot \vec{S}_{i+1} = \sum_i \left[S_i^z S_{i+1}^z + \frac{1}{2} \left(S_i^\dagger S_{i+1}^- + S_i^- S_{i+1}^\dagger \right) \right]$$

where $\vec{S}_i = \vec{\sigma}_i/2$ are the spin operators, σ_i are the Pauli matrices, and $S_i^\pm = S_i^x \pm i S_i^y$.

2.2.1 Exercise

Calculate the ground state energy of the antiferromagnetic Heisenberg model for a chain of spins $S = 1/2$ using the finite version of the DMRG algorithm. You should be able to pass the number of sites of the chain, the number of states kept during the DMRG truncation, and the number of sweeps during the finite algorithm as parameters. The output should be the energy per site, entanglement entropy and, truncation error at each step of the algorithm. You could:

- make a plot of the energy per site versus system size to see how converges. The relative error in energy is given by the truncation error. In a given sweep, where is the energy best approximate to the actual value?
- for the same size you chose in the last exercise, repeat the same extrapolation for the energy to zero truncation error. Compare this with the exact value for a finite size given by the Bethe Anstatz, and with the results you
- find the scaling of the entanglement entropy for the Heisenberg model. The entanglement entropy for open boundary conditions scales as $S(x) = \frac{c}{6} \log \left[\frac{2L}{\pi} sin \left(\frac{\pi x}{L} \right) \right]$, where c, x, L are the central charge of the model; the size of the part of the chain you keep in the reduced density matrix, and the size of the whole chain, respectively.

2.2.2 Solution

This is the first implementation of a full DMRG algorithm in the tutorial. The full DMRG algorithm involves doing first the infinite version of the DMRG algorithm, which we cover in the last exercise, and then a number of sweeps

keeping the size of the system fixed. The latter are called finite algorithm sweeps. During the finite sweeps, one block is growing exactly as during the infinite version of the algorithm, while the other has to shrink to keep the system size constant. As the truncation in DMRG can take you only for larger block sizes, you have to use an old version of the block that is shrinking. This is done saving each of the block when they grow, so you simply pull the right one from a list of old blocks.

To being able to reuse this code latter we are going to put all the stuff related to the AF Heisenberg model in a new object. In this way it's easier to switch between different models. This bring nothing new to the algorithm itself. The file with the model is something like this:

```
"""A few convenience functions to setup the Heisenberg model.

.. math::
    H=\sum_{i}\vec{S}_{i}\cdot\vec{S}_{i+1}=
    \sum_{i}\left[S^{z}_{i}S^{z}_{i+1}+
    \frac{1}{2}\left(S^{\dagger}_{i}S^{-}_{i+1}+
    S^{-}_{i}S^{\dagger}_{i+1}\right)\right]

"""
class HeisenbergModel(object):
    """Implements a few convenience functions for AF Heisenberg.

    Does exactly that.
    """
    def __init__(self):
        super(HeisenbergModel, self).__init__()

    def set_hamiltonian(self, system):
        """Sets a system Hamiltonian to the AF Heisenberg Hamiltonian.

        Does exactly this. If the system hamiltonian has some other terms on
        it, there are not touched. So be sure to use this function only in
        newly created `System` objects.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonain for.
        """
        system.clear_hamiltonian()
        if 'bh' in system.left_block.operators.keys():
            system.add_to_hamiltonian(left_block_op='bh')
        if 'bh' in system.right_block.operators.keys():
            system.add_to_hamiltonian(right_block_op='bh')
        system.add_to_hamiltonian('id', 'id', 's_z', 's_z')
        system.add_to_hamiltonian('id', 'id', 's_p', 's_m', .5)
        system.add_to_hamiltonian('id', 'id', 's_m', 's_p', .5)
        system.add_to_hamiltonian('id', 's_z', 's_z', 'id')
        system.add_to_hamiltonian('id', 's_p', 's_m', 'id', .5)
        system.add_to_hamiltonian('id', 's_m', 's_p', 'id', .5)
        system.add_to_hamiltonian('s_z', 's_z', 'id', 'id')
        system.add_to_hamiltonian('s_p', 's_m', 'id', 'id', .5)
        system.add_to_hamiltonian('s_m', 's_p', 'id', 'id', .5)

    def set_block_hamiltonian(self, tmp_matrix_for_bh, system):
        """Sets the block Hamiltonian to be what you need for AF Heisenberg.

        Parameters
        ----------
```

```
        tmp_matrix_for_bh : a numpy array of ndim = 2.
            An auxiliary matrix to keep track of the result.
        system : a System.
            The System you want to set the Hamiltonian for.
        """
        # If you have a block hamiltonian in your block, add it
        if 'bh' in system.growing_block.operators.keys():
            system.add_to_block_hamiltonian(tmp_matrix_for_bh, 'bh', 'id')
        system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_z', 's_z')
        system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_p', 's_m', .5)
        system.add_to_block_hamiltonian(tmp_matrix_for_bh, 's_m', 's_p', .5)

    def set_operators_to_update(self, system):
        """Sets the operators to update to be what you need to AF Heisenberg.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.

        Notes
        -----
        The block Hamiltonian, althought needs to be updated, is treated
        separately by the very functions in the `System` class.
        """
        system.add_to_operators_to_update('s_z', site_op='s_z')
        system.add_to_operators_to_update('s_p', site_op='s_p')
        system.add_to_operators_to_update('s_m', site_op='s_m')
```

The real changes are in the functions to grow the blocks, and perform the infinite and finite DMRG steps. As was said above these functions are included not in the main file, as in the previous exercise, but in the *System* class. Apart from the minor changes introduced by this change in the interface, you show pay attention to the changes in the functions themselves.

The first change we will make is to grow the system asymmetrically. This means that during the infinite version of the algorithm we keep on block (the right one) one-site long. You do this simply but not growing the right block:

Next thing is to write the function to implement the DMRG step during the finite algorithm. The only difference with the infinite version is that now in addition to grow one of the blocks, you set the other one to be the one with the proper size in a previous sweep.

During the finite sweeps you want to increase the number of states that you keep. A good way to do that is increasing them linearly at each half-sweep from the number of states at the end of the infinite algorithm to the number you want to keep at the end.

The rest is just doing a loop for the sweeps, each finite sweep comprising a "half-sweep" to the left and a "half-sweep" to the right, and being sure that during the finite algorithm the size of the system is constant. The implementation (with some extras for saving the results and the rest) looks like this:

```
def main(args):
    #
    # create a system object with spin one-half sites and blocks, and set
    # its model to be the TFIM.
    #
    spin_one_half_site = SpinOneHalfSite()
    system = System(spin_one_half_site)
    system.model = HeisenbergModel()
    #
    # read command-line arguments and initialize some stuff
```

```
    #
    number_of_sites = int(args['-n'])
    number_of_states_kept = int(args['-m'])
    number_of_sweeps = int(args['-s'])
    number_of_states_infinite_algorithm = 10
    if number_of_states_kept < number_of_states_infinite_algorithm:
        number_of_states_kept = number_of_states_infinite_algorithm
    sizes = []
    energies = []
    entropies = []
    truncation_errors = []
    system.number_of_sites = number_of_sites
    #
    # infinite DMRG algorithm
    #
    max_left_block_size = number_of_sites - 3
    for left_block_size in range(1, max_left_block_size+1):
        energy, entropy, truncation_error = (
            system.infinite_dmrg_step(left_block_size,
                                      number_of_states_infinite_algorithm) )
        current_size = left_block_size + 3
        sizes.append(left_block_size)
        energies.append(energy)
        entropies.append(entropy)
        truncation_errors.append(truncation_error)
    #
    # finite DMRG algorithm
    #
    states_to_keep = calculate_states_to_keep(number_of_states_infinite_algorithm,
                                              number_of_states_kept,
                                              number_of_sweeps)
    half_sweep = 0
    while half_sweep < len(states_to_keep):
        # sweep to the left
        for left_block_size in range(max_left_block_size, 0, -1):
            states = states_to_keep[half_sweep]
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('right', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
        # sweep to the right
        # if this is the last sweep, stop at the middle
        if half_sweep == 2 * number_of_sweeps - 1:
            max_left_block_size = number_of_sites / 2 - 1
        for left_block_size in range(1, max_left_block_size + 1):
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('left', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
    #
    # save results
    #
```

```
    output_file = os.path.join(os.path.abspath(args['--dir']), args['--output'])
    f = open(output_file, 'w')
    zipped = zip (sizes, energies, entropies, truncation_errors)
    f.write('\n'.join('%s %s %s %s' % x for x in zipped))
    f.close()
    print 'Results stored in ' + output_file
```

See a full implementation of the above code. To learn how to run this code you can use:

```
$ ./tutorial/solutions/heisenberg.py --help
Implements the full DMRG algorithm for the S=1/2 AF Heisenberg.

Calculates the ground state energy and wavefunction for the
antiferromagnetic Heisenberg model for a chain of spin one-half. The
calculation of the ground state is done using the full DMRG algorithm,
i.e. first the infinite algorithm, and then doing sweeps for convergence
with the finite algorithm.

Usage:
  heisenberg.py (-m=<states> -n=<sites> -s=<sweeps>) [--dir=DIR -o=FILE]
  heisenberg.py -h | --help

  Options:
    -h --help         Shows this screen.
    -n <sites>        Number of sites of the chain.
    -m <states>       Number of states kept.
    -s <sweeps>       Number of sweeps in the finite algorithm.
    -o --output=FILE  Ouput file [default: heisenberg.dat]
    --dir=DIR         Ouput directory [default: ./]
```

You can use this script to plot the entropies and fit them to the result from conformal field theory.

2.3 DMRG algorithm for the Ising model in a transverse field

The goal of this exercise is to implement the finite version of the DMRG algorithm for the Ising model in a transverse field (TFIM) for a chain of spins one-half and study the model close to the quantum phase transition.

The Hamiltonian is:

$$H = \sum_i \left(-J S_i^z S_{i+1}^z - h S_i^x \right)$$

where $\vec{S}_i = \vec{\sigma}_i / 2$ are the spin operators, and σ_i are the Pauli matrices.

This model has a quantum phase transition at $h/J = 1.0$. At the critical point the exact value for the ground state energy for a finite system with open boundary conditions is given by:

$$E/J = 1 - cosec\left(\frac{\pi}{2\,(2L+1)} \right)$$

The high field phase $h/J > 1.0$ is a paramagnet with an order parameter $\langle S^x \rangle \neq 0$. The low field phase $h/J < 1.0$ is a ferromagnet with an order parameter $\langle S^z \rangle \neq 0$. At the critical point, both order parameters go to zero.

The exact values of the energies at the critical point are:

L	E/L (J)
16	-1.2510242438
20	-1.255389856
32	-1.2620097863
40	-1.264235845
64	-1.267593439

2.3.1 Exercise

Study the quantum phase transition in the TFIM by measuring the order parameters at each side of the transition and comparing the energy at the critical point with the exact result. The critical point is

- calculate the energy per site for a given system size and different number of states kept, make a plot, and extrapolate the values of the energy to zero truncation error with a linear fit. Compare this with the exact value for a finite size given by the exact solution.
- measure the value of the order parameters for a given system size and a few values of h/J around the critical point. Use a reasonable system size and a number of states, so you actually are able to get to sample the phase diagram around the critical point. Get the order parameter by summing up the values of S^x or S^z for all sites of the chain. Plot both order parameters versus h/J.

2.3.2 Solution

The implementation goes is pretty similar to the one for the Heisenberg model of the last exercise. The main change is of course the change of the Hamiltonian, block Hamiltonian, and the operators you need to update after each DMRG step.

To simplify things and switching between different models, the *System* class has a few convience functions to do grow the blocks, perform the infinite and finite DMRG steps, and set the Hamiltonian. These are the same functions we have seen before, but now written as methods of the *System* class.

When using these methods you have to write a *Model* class that contains the details of the TFIM model:

```
class TranverseFieldIsingModel(object):
    """Implements a few convenience functions for the TFIM.

    Does exactly that.
    """
    def __init__(self, h = 0):
        super(TranverseFieldIsingModel, self).__init__()
        self.h = h

    def set_hamiltonian(self, system):
        """Sets a system Hamiltonian to the TFIM Hamiltonian.

        Does exactly this. If the system hamiltonian has some other terms on
        it, there are not touched. So be sure to use this function only in
        newly created `System` objects.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonain for.
        """
        system.clear_hamiltonian()
        if 'bh' in system.left_block.operators.keys():
```

```
            system.add_to_hamiltonian(left_block_op='bh')
        if 'bh' in system.right_block.operators.keys():
            system.add_to_hamiltonian(right_block_op='bh')
        system.add_to_hamiltonian('id', 'id', 's_z', 's_z')
        system.add_to_hamiltonian('id', 's_z', 's_z', 'id')
        system.add_to_hamiltonian('s_z', 's_z', 'id', 'id')
        system.add_to_hamiltonian('id', 'id', 'id', 's_x', self.h)
        system.add_to_hamiltonian('id', 'id', 's_x', 'id', self.h)
        system.add_to_hamiltonian('id', 's_x', 'id', 'id', self.h)
        system.add_to_hamiltonian('s_x', 'id', 'id', 'id', self.h)

    def set_block_hamiltonian(self, system):
        """Sets the block Hamiltonian to be what you need for TFIM.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.
        """
        # If you have a block hamiltonian in your block, add it
        if 'bh' in system.growing_block.operators.keys():
            system.add_to_block_hamiltonian('bh', 'id')
        system.add_to_block_hamiltonian('s_z', 's_z')
        system.add_to_hamiltonian('id', 's_x', self.h)
        system.add_to_hamiltonian('s_x', 'id', self.h)

    def set_operators_to_update(self, system):
        """Sets the operators to update to be what you need to TFIM.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.
        """
        # If you have a block hamiltonian in your block, update it
        if 'bh' in system.growing_block.operators.keys():
            system.add_to_operators_to_update('bh', block_op='bh')
        system.add_to_operators_to_update('s_z', site_op='s_z')
        system.add_to_operators_to_update('s_x', site_op='s_x')
```

The best thing is to look at the final implementation and compare to the previous one for the Heisenberg model:

```
def main(args):
    #
    # create a system object with spin one-half sites and blocks, and set
    # its model to be the TFIM.
    #
    spin_one_half_site = SpinOneHalfSite()
    system = System(spin_one_half_site)
    system.model = TranverseFieldIsingModel()
    #
    # read command-line arguments and initialize some stuff
    #
    number_of_sites = int(args['-n'])
    number_of_states_kept = int(args['-m'])
    number_of_sweeps = int(args['-s'])
    system.model.H = float(args['-H'])
    number_of_states_infinite_algorithm = 10
```

```
    if number_of_states_kept < number_of_states_infinite_algorithm:
        number_of_states_kept = number_of_states_infinite_algorithm
    sizes = []
    energies = []
    entropies = []
    truncation_errors = []
    system.number_of_sites = number_of_sites
    #
    # infinite DMRG algorithm
    #
    max_left_block_size = number_of_sites - 3
    for left_block_size in range(1, max_left_block_size+1):
        energy, entropy, truncation_error = (
            system.infinite_dmrg_step(left_block_size,
                                      number_of_states_infinite_algorithm) )
        current_size = left_block_size + 3
        sizes.append(left_block_size)
        energies.append(energy)
        entropies.append(entropy)
        truncation_errors.append(truncation_error)
    #
    # finite DMRG algorithm
    #
    states_to_keep = calculate_states_to_keep(number_of_states_infinite_algorithm,
                                              number_of_states_kept,
                                              number_of_sweeps)
    half_sweep = 0
    while half_sweep < len(states_to_keep):
        # sweep to the left
        for left_block_size in range(max_left_block_size, 0, -1):
            states = states_to_keep[half_sweep]
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('right', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
        # sweep to the right
        # if this is the last sweep, stop at the middle
        if half_sweep == 2 * number_of_sweeps - 1:
            max_left_block_size = number_of_sites / 2 - 1
        for left_block_size in range(1, max_left_block_size + 1):
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('left', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
    #
    # save results
    #
    output_file = os.path.join(os.path.abspath(args['--dir']), args['--output'])
    f = open(output_file, 'w')
    zipped = zip (sizes, energies, entropies, truncation_errors)
    f.write('\n'.join('%s %s %s %s' % x for x in zipped))
    f.close()
```

```
    print 'Results stored in ' + output_file
```

See a full implementation of the above code. To learn how to run this code you can use:

```
$ ./solutions/tfim.py --help
Implements the full DMRG algorithm for the S=1/2 TFIM.

Calculates the ground state energy and wavefunction for the
Ising model in a transverse field for a chain of spin one-half. The
calculation of the ground state is done using the full DMRG algorithm,
i.e. first the infinite algorithm, and then doing sweeps for
convergence with the finite algorithm.

Usage:
  tfim.py (-m=<states> -n=<sites> -s=<sweeps> -H=<field>) [--dir=DIR -o=FILE]
  tfim.py -h | --help

Options:
  -h --help         Shows this screen.
  -n <sites>        Number of sites of the chain.
  -m <states>       Number of states kept.
  -s <sweeps>       Number of sweeps in the finite algorithm.
  -H <field>        Magnetic field in units of coupling between spins.
  -o --output=FILE  Ouput file [default: tfim.dat]
  --dir=DIR         Ouput directory [default: ./]
```

CHAPTER 3

Electronic systems

3.1 DMRG algorithm for the Hubbard model

The goal of this exercise is to implement the finite version of the DMRG algorithm for the one-dimensional Hubbard model. The Hamiltonian is:

$$H = -t \sum_{i,\sigma} \left(c^{\dagger}_{i,\sigma} c_{i,\sigma} + h.c. \right) + U \sum_{i} n_{i,\uparrow} n_{i,\downarrow}$$

where $c_{i,\sigma}$ is the destruction operator for an electron at site i and spin σ, and $n_{i,\sigma} = c^{\dagger}_{i,\sigma} c_{i,\sigma}$.

3.1.1 Exercise

Decide by yourself what you want to calculate with this code.

3.1.2 Solution

The implementation is straighforward now. You have to write a model class which with the functions to set the Hamiltonian, block Hamiltonians, and operators to update. A possible implementation is:

```
class HubbardModel(object):
    """Implements a few convenience functions for Hubbard model.

    Does exactly that.
    """
    def __init__(self):
        super(HubbardModel, self).__init__()

    def set_hamiltonian(self, system):
        """Sets a system Hamiltonian to the Hubbard Hamiltonian.

        Does exactly this. If the system hamiltonian has some other terms on
        it, there are not touched. So be sure to use this function only in
        newly created `System` objects.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.
        """
```

```
        system.clear_hamiltonian()
        if 'bh' in system.left_block.operators.keys():
            system.add_to_hamiltonian(left_block_op='bh')
        if 'bh' in system.right_block.operators.keys():
            system.add_to_hamiltonian(right_block_op='bh')
        system.add_to_hamiltonian('c_up', 'c_up_dag', 'id', 'id', -1.)
        system.add_to_hamiltonian('c_up_dag', 'c_up', 'id', 'id', -1.)
        system.add_to_hamiltonian('c_down', 'c_down_dag', 'id', 'id', -1.)
        system.add_to_hamiltonian('c_down_dag', 'c_down', 'id', 'id', -1.)
        system.add_to_hamiltonian('id', 'c_up', 'c_up_dag', 'id', -1.)
        system.add_to_hamiltonian('id', 'c_up_dag', 'c_up', 'id', -1.)
        system.add_to_hamiltonian('id', 'c_down', 'c_down_dag', 'id', -1.)
        system.add_to_hamiltonian('id', 'c_down_dag', 'c_down', 'id', -1.)
        system.add_to_hamiltonian('id', 'id', 'c_up', 'c_up_dag', -1.)
        system.add_to_hamiltonian('id', 'id', 'c_up_dag', 'c_up', -1.)
        system.add_to_hamiltonian('id', 'id', 'c_down', 'c_down_dag', -1.)
        system.add_to_hamiltonian('id', 'id', 'c_down_dag', 'c_down', -1.)
        system.add_to_hamiltonian('u', 'id', 'id', 'id', self.U)
        system.add_to_hamiltonian('id', 'u', 'id', 'id', self.U)
        system.add_to_hamiltonian('id', 'id', 'u', 'id', self.U)
        system.add_to_hamiltonian('id', 'id', 'id', 'u', self.U)

    def set_block_hamiltonian(self, system):
        """Sets the block Hamiltonian to the Hubbard model block Hamiltonian.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.
        """
        # If you have a block hamiltonian in your block, add it
        if 'bh' in system.growing_block.operators.keys():
            system.add_to_block_hamiltonian('bh', 'id')
        system.add_to_block_hamiltonian('c_up', 'c_up_dag', -1.)
        system.add_to_block_hamiltonian('c_up_dag', 'c_up', -1.)
        system.add_to_block_hamiltonian('c_down', 'c_down_dag', -1.)
        system.add_to_block_hamiltonian('c_down_dag', 'c_down', -1.)
        system.add_to_block_hamiltonian('id', 'u', self.U)
        system.add_to_block_hamiltonian('u', 'id', self.U)

    def set_operators_to_update(self, system):
        """Sets the operators to update to the ones for the Hubbard model.

        Parameters
        ----------
        system : a System.
            The System you want to set the Hamiltonian for.
        """
        # If you have a block hamiltonian in your block, update it
        if 'bh' in system.growing_block.operators.keys():
            system.add_to_operators_to_update('bh', block_op='bh')
        system.add_to_operators_to_update('c_up', site_op='c_up')
        system.add_to_operators_to_update('c_up_dag', site_op='c_up_dag')
        system.add_to_operators_to_downdate('c_down', site_op='c_down')
        system.add_to_operators_to_downdate('c_down_dag', site_op='c_down_dag')
        system.add_to_operators_to_update('u', site_op='u')
```

You also have to write a single site class for the electronic site, providing operators defined in the Hilbert space of the

site:

```
class ElectronicSite(Site):
    """A site for electronic models

    You use this site for models where the single sites are electron
    sites. The Hilbert space is ordered such as:

    - the first state, labelled 0,  is the empty site,
    - the second, labelled 1, is spin down,
    - the third, labelled 2, is spin up, and
    - the fourth, labelled 3, is double occupancy.

    Notes
    -----
    Postcond: The site has already built-in the spin operators for:

    - c_up : destroys an spin up electron,
    - c_up_dag, creates an spin up electron,
    - c_down, destroys an spin down electron,
    - c_down_dag, creates an spin down electron,
    - s_z, component z of spin,
    - s_p, raises the component z of spin,
    - s_m, lowers the component z of spin,
    - n_up, number of electrons with spin up,
    - n_down, number of electrons with spin down,
    - n, number of electrons, i.e. n_up+n_down, and
    - u, number of double occupancies, i.e. n_up*n_down.

    """
    def __init__(self):
        super(ElectronicSite, self).__init__(4)
        # add the operators
        self.add_operator("c_up")
        self.add_operator("c_up_dag")
        self.add_operator("c_down")
        self.add_operator("c_down_dag")
        self.add_operator("s_z")
        self.add_operator("s_p")
        self.add_operator("s_m")
        self.add_operator("n_up")
        self.add_operator("n_down")
        self.add_operator("n")
        self.add_operator("u")
        # for clarity
        c_up = self.operators["c_up"]
        c_up_dag = self.operators["c_up_dag"]
        c_down = self.operators["c_down"]
        c_down_dag = self.operators["c_down_dag"]
        s_z = self.operators["s_z"]
        s_p = self.operators["s_p"]
        s_m = self.operators["s_m"]
        n_up = self.operators["n_up"]
        n_down = self.operators["n_down"]
        n = self.operators["n"]
        u = self.operators["u"]
        # set the matrix elements different from zero to the right values
        # TODO: missing s_p, s_m
        c_up[0,2] = 1.0
```

```
        c_up[1,3] = 1.0
        c_up_dag[2,0] = 1.0
        c_up_dag[3,1] = 1.0
        c_down[0,1] = 1.0
        c_down[2,3] = 1.0
        c_down_dag[1,0] = 1.0
        c_down_dag[3,2] = 1.0
        s_z[1,1] = -1.0
        s_z[2,2] = 1.0
        n_up[2,2] = 1.0
        n_up[3,3] = 1.0
        n_down[1,1] = 1.0
        n_down[3,3] = 1.0
        n[1,1] = 1.0
        n[2,2] = 1.0
        n[3,3] = 2.0
        u[3,3] = 1.0
```

The implementation of the DMRG algorithm itself has only minor changes with respect what you have done before:

```
def main(args):
    #
    # create a system object with electron sites and blocks, and set
    # its model to be the Hubbard model.
    #
    electronic_site = ElectronicSite()
    system = System(electronic_site)
    system.model = HubbardModel()
    #
    # read command-line arguments and initialize some stuff
    #
    number_of_sites = int(args['-n'])
    number_of_states_kept = int(args['-m'])
    number_of_sweeps = int(args['-s'])
    system.model.U = float(args['-U'])
    number_of_states_infinite_algorithm = 10
    if number_of_states_kept < number_of_states_infinite_algorithm:
        number_of_states_kept = number_of_states_infinite_algorithm
    sizes = []
    energies = []
    entropies = []
    truncation_errors = []
    system.number_of_sites = number_of_sites
    #
    # infinite DMRG algorithm
    #
    max_left_block_size = number_of_sites - 3
    for left_block_size in range(1, max_left_block_size + 1):
        energy, entropy, truncation_error = (
            system.infinite_dmrg_step(left_block_size,
                                      number_of_states_infinite_algorithm) )
        current_size = left_block_size + 3
        sizes.append(left_block_size)
        energies.append(energy)
        entropies.append(entropy)
        truncation_errors.append(truncation_error)
    #
    # finite DMRG algorithm
    #
```

```
    states_to_keep = calculate_states_to_keep(number_of_states_infinite_algorithm,
                                              number_of_states_kept,
                                              number_of_sweeps)
    half_sweep = 0
    while half_sweep < len(states_to_keep):
        # sweep to the left
        for left_block_size in range(max_left_block_size, 0, -1):
            states = states_to_keep[half_sweep]
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('right', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
        # sweep to the right
        # if this is the last sweep, stop at the middle
        if half_sweep == 2 * number_of_sweeps - 1:
            max_left_block_size = number_of_sites / 2 - 1
        for left_block_size in range(1, max_left_block_size + 1):
            energy, entropy, truncation_error = (
                system.finite_dmrg_step('left', left_block_size, states) )
            sizes.append(left_block_size)
            energies.append(energy)
            entropies.append(entropy)
            truncation_errors.append(truncation_error)
        half_sweep += 1
    #
    # save results
    #
    output_file = os.path.join(os.path.abspath(args['--dir']), args['--output'])
    f = open(output_file, 'w')
    zipped = zip (sizes, energies, entropies, truncation_errors)
    f.write('\n'.join('%s %s %s %s' % x for x in zipped))
    f.close()
    print 'Results stored in ' + output_file
```

See `a full implementation of the above code`. To learn how to run this code you can use:

```
$ ./solutions/tfim.py --help
Implements the full DMRG algorithm for the S=1/2 TFIM.

Calculates the ground state energy and wavefunction for the
Ising model in a transverse field for a chain of spin one-half. The
calculation of the ground state is done using the full DMRG algorithm,
i.e. first the infinite algorithm, and then doing sweeps for
convergence with the finite algorithm.

Usage:
  tfim.py (-m=<states> -n=<sites> -s=<sweeps> -H=<field>) [--dir=DIR -o=FILE]
  tfim.py -h | --help

Options:
  -h --help         Shows this screen.
  -n <sites>        Number of sites of the chain.
  -m <states>       Number of states kept.
  -s <sweeps>       Number of sweeps in the finite algorithm.
  -H <field>        Magnetic field in units of coupling between spins.
```

```
  -o --output=FILE  Ouput file [default: tfim.dat]
  --dir=DIR         Ouput directory [default: ./]
```