Hubbard Model - Part 3, QuTiP code

November 24, 2015

```
In [72]: from jupyter_core.paths import jupyter_config_dir, jupyter_data_dir

In [73]: jupyter_data_dir()

Out[73]: '/Users/deepak/Library/Jupyter'

In [3]: import numpy as np
        import matplotlib.pyplot as plt
        from qutip import *
        %matplotlib inline
```

0.1 Hubbard Model

0.1.1 Hamiltonian

$$H_h = -t \sum_{<i,j>\sigma} c^\dagger_{i\sigma} c_{j\sigma} + U \sum_i n_{i\uparrow} n_{j\downarrow} - \mu \sum_i (n_{i\uparrow} + n_{i\downarrow})$$

where the first term is the kinetic energy. The sum is over both possible spin at each lattice site $\sigma \in \{\uparrow, \downarrow\}$. Second term is the potential energy due to repulsion of electrons at a site containing two electrons. The last term is the chemical potential associated with adding particles to the system.

The partition function for a system in a thermal state is given by:

$$Z = \mathrm{Tr}\left[e^{-\beta H}\right] = \sum_\alpha \langle\alpha| e^{-\beta H} |\alpha\rangle$$

0.1.2 Partition Function

For $H\_\{h\}$ on a single site, the partition function becomes:

$$Z_h = 1 + e^{\beta(\mu+t)} + e^{\beta(2t+2\mu-U)}$$

If we redefine the chemical potential $\mu \to \mu + U/2$, then the Hubbard hamiltonian becomes:

$$H = -t \sum_{<i,j>\sigma} c^\dagger_{i\sigma} c_{j\sigma} + U \sum_i \left(n_{i\uparrow} - \frac{1}{2}\right)\left(n_{j\downarrow} - \frac{1}{2}\right) - \mu \sum_i (n_{i\uparrow} + n_{i\downarrow})$$

0.1.3 Energy

The energy for the single site Hubbard model is:

$$E = \langle H + \mu n \rangle = \mathrm{Tr}\left[(H + \mu n) e^{-\beta H}\right] \tag{1}$$

$$= \frac{1}{Z} \sum_\alpha \langle\alpha| (H + \mu n) e^{-\beta H} |\alpha\rangle \tag{2}$$

$$= \frac{U e^{2\beta(t+\mu-U/2)}}{1 + e^{\beta(\mu+t)} + e^{\beta(2t+2\mu-U)}} \tag{3}$$

0.1.4 Occupation Number

and the occupation number is:

$$\rho = \langle n \rangle = \mathrm{Tr}\left[ne^{-\beta H}\right] \tag{4}$$

$$= \frac{2e^{\beta(\mu+t)} + 2e^{2\beta(t+\mu-U/2)}}{1 + e^{\beta(\mu+t)} + e^{\beta(2t+2\mu-U)}} \tag{5}$$

0.2 Operators and States in QuTiP

Rather than using the analytical expression for the partition function, energy and other properties of the one-site Hubbard model, we will now use the Python package QuTiP to define the Hubbard model Hamiltonian on N sites. For this we will need to define creation, annihilation and number operators.

For a single site system a creation operator is simply defined using the `create()` function:

$$\mathrm{create}(2) \Rightarrow c^\dagger$$

which returns the matrix:

$$\begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \end{pmatrix}$$

```
In [4]: create(2)
```

Out[4]:

Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \end{pmatrix}$$

0.2.1 Operators for many-body systems

When we have more than one site, the operator must be defined as an operator acting of the full Hilbert space of the system:

$$\mathcal{H} = \otimes_{i=1}^{N} \mathcal{H}_i$$

where $\mathcal{H}_i$ is the Hilbert space corresponding to a single site. Thus the creation operator for the j^{th} site is given by:

$$\mathbf{1}_1 \otimes \ldots \mathbf{1}_{j-1} \otimes c_j^\dagger \otimes \mathbf{1}_{j+1} \ldots \mathbf{1}_N$$

where $\mathbf{1}_i$ is the identity operator acting on the i^{th} site.

The code for this is as follows:

Utility Functions

```
In [5]: def identityList(N = 1,dims = 2):
            '''Returns a list of N identity operators for a N site spin-system with a
            Hilber space at each state of dimensionality dims
            '''

            iden = identity(dims)

            iden_list = []

            [iden_list.append(iden) for i in range(N)]

            return iden_list
```

Position Representation Operators

```
In [6]: def posOperatorN(oper, i = 0, N = 1):
            '''Returns the operator given by oper, in the position representation, for the i^th site of
            with a Hilbert space of dimensionality dims at each site'''

            if not isinstance(oper, Qobj):
                raise TypeError('oper must of type qutip.Qobj')

            if not oper.isoper:
                raise ValueError('oper must be a qutip operator')

            shape = oper.shape

            if shape[0] == shape[1]:
                dims = shape[0]
            else:
                raise ValueError('oper must be a square matrix')

            if oper == identity(oper.shape[0]):
                return tensor(identityList(N,oper.shape[0]))
            else:
                iden_list = identityList(N, oper.shape[0])
                iden_list[i] = oper
                return tensor(iden_list)

In [7]: def posCreationOpN(i=0, N=10):
            '''Returns the creation operator in the position representation for the i^th site of an N s
            with a Hilbert space of dimensionality dims at each site'''

            return posOperatorN(create(2),i,N)

In [8]: def posDestructionOpN(i=0, N=10):
            '''Returns the destruction operator in the position representation for the i^th site of an
            with a Hilbert space of dimensionality dims at each site'''

            return posOperatorN(destroy(2),i,N)

In [9]: def posOperHamiltonian(oper, N = 10, coefs = [1]*10, dims=2):
            ''' Returns the Hamiltonian, in position representation, given by the sum of oper acting on
            with a weight given by the values in coefs
            '''

            if not isinstance(oper, Qobj):
                raise ValueError('oper must be of type Qobj')
            else:
                if not oper.isoper:
                    raise ValueError('oper must be an operator')

            H = 0

            for i in range(N):
                op_list = identityList(N, dims)
                op_list[i] = oper
                H += coefs[i]*tensor(op_list)
```

```
            return H
```

In [11]:
```
# To get the number operator at the 3rd site, in a N=5 site system, we do the following
numOp = create(2)*destroy(2)
posOperatorN(sigmaz(), N=5, i=4)
```

Out[11]:

Quantum object: dims = [[2, 2, 2, 2, 2], [2, 2, 2, 2, 2]], shape = [32, 32], type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 \end{pmatrix}$$

0.3 Unitary Transformations

In general, given any unitary matrix U, which transforms from one set of basis vectors $\{|\psi_i\rangle\}$ to another set $\{|\phi_i\rangle\}$:

$$|\phi_i\rangle = U_{ij}|\psi_j\rangle$$

the corresponding action of U on the space of operators is given by:

$$\mathcal{O} \to U^{-1}\mathcal{O}U$$

or in terms of indices:

$$\mathcal{O}_{ij} = U^{-1}_{ik}\mathcal{O}_{kl}U_{lj}$$

0.4 Momentum Representation

We transform operators to momentum space as follows:

$$c^\dagger_{\mathbf{k}\sigma} = \frac{1}{\sqrt{N}} \sum_{\mathbf{l}} e^{i\mathbf{k}\cdot\mathbf{l}} c^\dagger_{\mathbf{l}\sigma}$$

with the inverse transformation being:

$$c^\dagger_{\mathbf{l}\sigma} = \frac{1}{\sqrt{N}} \sum_{\mathbf{k}} e^{-i\mathbf{k}\cdot\mathbf{l}} c^\dagger_{\mathbf{k}\sigma}$$

where we have used the orthogonality condition:

$$\frac{1}{N} \sum_{\mathbf{l}} e^{-i(\mathbf{k}-\mathbf{k}')\cdot\mathbf{l}} = \delta(\mathbf{k}-\mathbf{k}')$$

This transformation can be written in terms of a matrix:

$$c^\dagger_{\mathbf{k}\sigma} = A_{\mathbf{k},\mathbf{l}}\, c^\dagger_{\mathbf{l}\sigma}$$

where

$$A_{\mathbf{k},\mathbf{l}} \equiv e^{i\mathbf{k}\cdot\mathbf{l}}$$

For a finite system, the momentum $\mathbf{k}$ and position $\mathbf{l}$ can take on values only in a finite set:

$$\mathbf{k} \in \{\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_n\}; \quad \mathbf{l} \in \{\mathbf{l}_1, \mathbf{l}_2, \ldots, \mathbf{l}_n\}$$

For a 1D system with N sites, $\mathbf{k}_n := k_n = 2\pi n/N$

More precisely, the momentum space operator $c^\dagger_{\mathbf{k}\sigma}$ can be written as the sum:

$$c^\dagger_{\mathbf{k}\sigma} = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} e^{i\mathbf{k}\cdot\mathbf{j}} \cdot \mathbf{1}_1 \otimes \ldots \mathbf{1}_{j-1} \otimes c^\dagger_{j\sigma} \otimes \mathbf{1}_{j+1} \ldots \mathbf{1}_N$$

The unitary matrix which transforms operators and states between position and momentum representations, is given by:

$$U_{\mathbf{k},\mathbf{l}} = \frac{1}{\sqrt{N}} e^{i\mathbf{k}\cdot\mathbf{l}}$$

The inverse of this matrix $U^\dagger \equiv U^{-1}$ transforms back from the momentum rep to the position rep.

$$U^{-1}_{\mathbf{k},\mathbf{l}} = U^\dagger_{\mathbf{k},\mathbf{l}} = U^\star_{\mathbf{l},\mathbf{k}}$$

The action of $U\{-1\}\_\{\mathbf{k},\mathbf{l}\}$ on $c^\dagger_{\mathbf{k}\sigma}$ is given by:

$$\sum_{k=1}^{N} U^{-1}_{\mathbf{k},\mathbf{l}} c^\dagger_{\mathbf{k}\sigma} = \sum_{k=1}^{N} U^\star_{\mathbf{l},\mathbf{k}} c^\dagger_{\mathbf{k}\sigma} = \frac{1}{N} \sum_{k,j=1}^{N} e^{-i\mathbf{l}\cdot\mathbf{k}} e^{i\mathbf{k}\cdot\mathbf{j}} \cdot \mathbf{1}_1 \otimes \ldots \mathbf{1}_{j-1} \otimes c^\dagger_{j\sigma} \otimes \mathbf{1}_{j+1} \ldots \mathbf{1}_N \tag{6}$$

$$= \sum_{j=1}^{N} \delta_{\mathbf{l},\mathbf{j}} \cdot \mathbf{1}_1 \otimes \ldots \mathbf{1}_{j-1} \otimes c^\dagger_{j\sigma} \otimes \mathbf{1}_{j+1} \ldots \mathbf{1}_N \tag{7}$$

$$= \mathbf{1}_1 \otimes \ldots \mathbf{1}_{l-1} \otimes c^\dagger_{l\sigma} \otimes \mathbf{1}_{l+1} \ldots \mathbf{1}_N \tag{8}$$

Position to Momentum Space The unitary matrix which transforms operators and states between position and momentum representations, is given by:

$$U_{\mathbf{k},\mathbf{l}} = e^{-i\mathbf{k}\cdot\mathbf{l}}$$

Given any operator $\mathcal{O}$ whose matrix elements in position space are $\mathcal{O}_{\mathbf{l},\mathbf{l}'}$, the matrix elements of the corresponding operator in momentum space are given by:

$$\mathcal{O}_{\mathbf{k},\mathbf{k}'} = U^{-1}_{\mathbf{k},\mathbf{l}} \mathcal{O}_{\mathbf{l},\mathbf{l}'} U_{\mathbf{l},\mathbf{k}}$$

The following code returns a QuTiP object corresponding to a matrix whose elements are $U_{\mathbf{k},\mathbf{l}}$.

Momentum Space Operators

```
In [16]: def posToMomentumOpN(oper, k = 0, N = 1):
             '''Returns the momentum space represenation of the operator given by oper for the k^th mom
             of an N site spin-chain with a Hilbert space of dimensionality dims at each site'''

             momOp = tensor(identityList(N,oper.shape[0]))
             invrtn = 1/np.sqrt(N)

         #    type(invrtn)

             for i in range(N):
                 momOp += invrtn * np.exp(1j*i*k) * posOperatorN(oper,i,N)

             return momOp
```

```
In [17]: def momCreationOpN(k = 0, N = 1):
             '''Returns the momentum space represenation of the creation operator for the k^th momentum
             of an N site spin-chain with a Hilbert space of dimensionality dims at each site'''

             return posToMomentumOpN(create(2),k,N)
```

```
In [18]: def momDestructionOpN(k = 0, N = 1):
             '''Returns the momentum space represenation of the operator given by oper for the k^th mom
             of an N site spin-chain with a Hilbert space of dimensionality dims at each site'''

             return qutip.dag(momCreationOpN(k,N))
```

```
In [19]: momCreationOpN(k=2,N=3)
```

Out[19]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = False

1.0	0.0	0.0	0.0	0.0	0.0	0.0	
$(-0.377-0.437j)$	1.0	0.0	0.0	0.0	0.0	0.0	
$(-0.240+0.525j)$	0.0	1.0	0.0	0.0	0.0	0.0	
0.0	$(-0.240+0.525j)$	$(-0.377-0.437j)$	1.0	0.0	0.0	0.0	
0.577	0.0	0.0	0.0	1.0	0.0	0.0	
0.0	0.577	0.0	0.0	$(-0.377-0.437j)$	1.0	0.0	
0.0	0.0	0.577	0.0	$(-0.240+0.525j)$	0.0	1.0	
0.0	0.0	0.0	0.577	0.0	$(-0.240+0.525j)$	$(-0.377-0.4$	

```
In [20]: momDestructionOpN(k=2,N=3)
```

Out[20]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = False

1.0	$(-0.377+0.437j)$	$(-0.240-0.525j)$	0.0	0.577	0.0	0.0	
0.0	1.0	0.0	$(-0.240-0.525j)$	0.0	0.577	0.0	
0.0	0.0	1.0	$(-0.377+0.437j)$	0.0	0.0	0.577	
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0
0.0	0.0	0.0	0.0	1.0	$(-0.377+0.437j)$	$(-0.240-0.525j)$	
0.0	0.0	0.0	0.0	0.0	1.0	0.0	$(-0.240$
0.0	0.0	0.0	0.0	0.0	0.0	1.0	$(-0.377$
0.0	0.0	0.0	0.0	0.0	0.0	0.0	

```
In [22]: _19.dag() - _20
```

Out[22]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Correct Interpretation of Indices It is important to keep in mind that the indices l, l' in the previous paragraph refer not to the

0.5 Model Hamiltonians

```
In [23]: def matrixPosToMom(N = 10):
             ''' Returns a QuTiP object corresponding to a matrix whose elements are:

             U_{k,l} = e^{-i*k*l}

             '''

             matrix = np.zeros([N,N],dtype=complex)

             for k in range(N):
                 for l in range(N):
                     matrix[k][l] = np.exp(1j*k*l)

             return Qobj(matrix)
```

```
In [24]: matrixPosToMom(N = 4)
```

Out[24]:

Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isherm = False

$$\begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & (0.540+0.841j) & (-0.416+0.909j) & (-0.990+0.141j) \\ 1.0 & (-0.416+0.909j) & (-0.654-0.757j) & (0.960-0.279j) \\ 1.0 & (-0.990+0.141j) & (0.960-0.279j) & (-0.911+0.412j) \end{pmatrix}$$

0.5.1 Hubbard Hamiltonian Code

Now at each site, one can have two electrons with spin up and down respectively. Thus our creation/annihilation operators also have a spin index σ:

$$\mathbf{1}_1 \otimes \ldots \mathbf{1}_{j-1} \otimes c^\dagger_{j\sigma} \otimes \mathbf{1}_{j+1} \ldots \mathbf{1}_N$$

In order to implement we need two sets of creation/annihilation operators. One set for up-spin and one set for down-spin.

```
In [27]: def hamiltonianHubbard(N = 10, t = 1, U = 1, mu = 0, periodic = True, shift = False, dims = 2)
             '''Returns operator corresponding to Hubbard Hamiltonian on N sites.
             Default value of N is 10. t, U and mu are the kinetic energy, potential energy and
             chemical potential respectively.
             If shift is False then first version of Hamiltonian is returned, else the second
             version (where the chemical potential is shifted £\mu \rightarrow \mu - U/2£) is
             returned.
             dims is the dimension of the Hilbert space for each electron. Default is 2'''

             # two sets of creation/destruction operators, labeled A and B.

             destroyA_list = []
             createA_list = []

             destroyB_list = []
             createB_list = []

             cOp = create(dims)
             dOp = destroy(dims)
```

```
    nOp = cOp * dOp

    nA_list = []
    nB_list = []

    idOp = identity(dims)

    idOp_list = []

    [idOp_list.append(idOp) for i in range(N)]

    superid = tensor(idOp_list) # identity operator for whole system

    H = 0

    for i in range(N):
        # Create list containing creation/destruction/number operators for each site

        createA_list.append(posOperatorN(cOp,N=N,i=i))
        createB_list.append(posOperatorN(cOp,N=N,i=i))

        destroyA_list.append(posOperatorN(dOp,N=N,i=i))
        destroyB_list.append(posOperatorN(dOp,N=N,i=i))

        nA_list.append(posOperatorN(nOp,N=N,i=i))
        nB_list.append(posOperatorN(nOp,N=N,i=i))

    if periodic == True:
        for i in range(N):
            H += - t * (createA_list[i%N] * destroyA_list[(i+1)%N] + createB_list[i%N] * destr
    else:
        for i in range(N-1):
            H += - t * (createA_list[i] * destroyA_list[i+1] + createB_list[i] * destroyB_list

    for i in range(N):
        H += - mu * (nA_list[i] + nB_list[i])
        if shift == True:
            H += U * (nA_list[i] - 0.5 * superid) * (nB_list[i] - 0.5 * superid)
        else:
            H += U * nA_list[i] * nB_list[i]

    return H
```

In [28]:
```
h1 = hamiltonianHubbard(mu=0.1,N=6,t=-1)
h1
```

Out[28]:

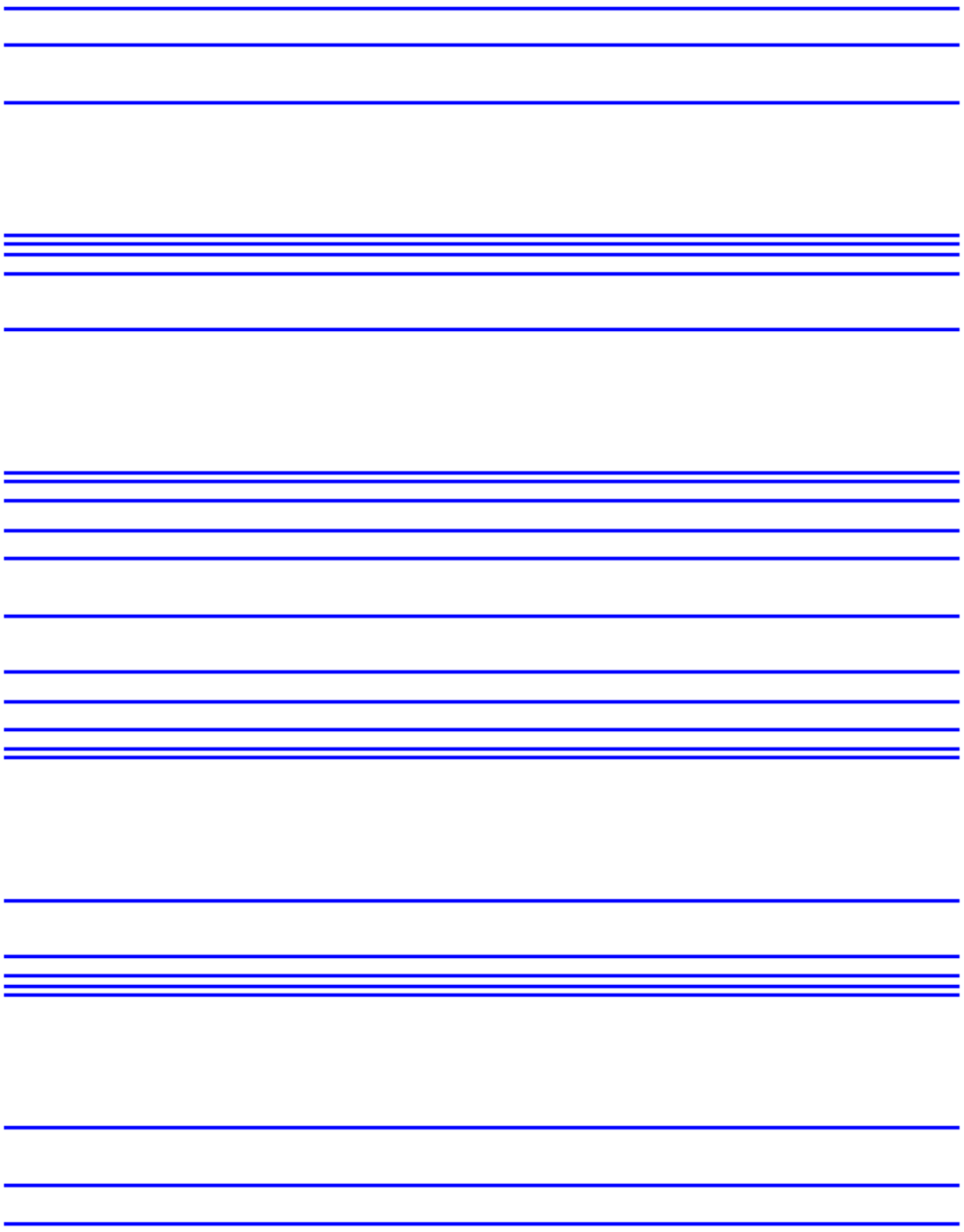
Quantum object: dims = [[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]], shape = [64, 64], type = oper, isherm = False

$$\begin{pmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.800 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.800 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.600 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 2.0 & 0.0 & 0.800 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 3.200 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 2.0 & 0.0 & 4.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 2.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 4.800
\end{pmatrix}$$

In [29]:
```
plot_energy_levels([_28])
```

```
/Users/deepak/anaconda/lib/python2.7/site-packages/numpy/core/numeric.py:462: ComplexWarning: Casting co
  return array(a, dtype, copy=False, order=order)
```

Out[29]:
```
(<matplotlib.figure.Figure at 0x107c97710>,
 <matplotlib.axes._subplots.AxesSubplot at 0x104558f90>)
```



0.5.2 Ising Hamiltonian Code

In [34]:
```
def hamiltonianIsing(N = 10, jcoefs = [], periodic = True, spin=0.5):
    '''Returns operator corresponding to Ising Hamiltonian for give spin on N sites.
    Default value of N is 10. jcoef is the coupling strength. Default is -1 for
    ferromagnetic interaction.
    Default value of spin is 0.5
    '''

    op_list = []

    jz = jmat(spin,'z')

    dimj = 2*spin + 1

    H = 0

    idlist = identityList(N, dimj)

    if len(jcoefs) == 0:
        jcoefs = [-1]*N

    for i in range(N):
        # Create list containing spin-z operators for each site:

        op_list.append(posOperatorN(jz,i=i,N=N))

    if periodic == True:
        for i in range(N):
            H += jcoefs[i%N]*op_list[i%N]*op_list[(i+1)%N]
    else:
        for i in range(N-1):
            H += jcoefs[i]*op_list[i]*op_list[i+1]

    return H
```

In [51]:
```
N = 5
#jcoefs = 2*np.random.random(N) - 1
jcoefs = [-1]*N
jcoefs
```

Out[51]:
```
[-1, -1, -1, -1, -1]
```

In [54]:
```
hamiltonianIsing(N,jcoefs, periodic = True)
```

Out[54]:

Quantum object: dims = [[2, 2, 2, 2, 2], [2, 2, 2, 2, 2]], shape = [32, 32], type = oper, isherm = True

$$\begin{pmatrix} -1.250 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -0.250 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -0.250 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -0.250 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -0.250 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -0.250 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -0.250 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -0.250 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & -0.250 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & -1.250 \end{pmatrix}$$

```
In [55]: plot_energy_levels([_54])
```

```
Out[55]: (<matplotlib.figure.Figure at 0x108fb8a10>,
          <matplotlib.axes._subplots.AxesSubplot at 0x108fa4750>)
```

0.5.3 Heisenberg Spin-Chain

operator corresponding to the Heisenberg 1D spin-chain on N sites.

$$H = -J \sum_{i=1}^{N} \mathbf{S}_n \cdot \mathbf{S}_{n+1}$$

where $ \mathbf{S}_n = (S_x, S_y, S_z) $ is the spin-operator acting on the n^{th} site.
for the n^{th} term in the sum, we have:

$$H_n = -J(S_n^x S_{n+1}^x + S_n^y S_{n+1}^y + S_n^z S_{n+1}^z)$$

S^x, S^y can be expressed in terms of the spin-flip operators S^+, S^-, as in:

$$S^x = \frac{1}{2}(S^+ + S^-); \qquad S^y = \frac{1}{2i}(S^+ - S^-)$$

Consequently the terms involving S^x, S^y in H_n take the form:

$$S_n^x S_{n+1}^x + S_n^y S_{n+1}^y = \frac{1}{4}(S_n^+ + S_n^-)(S_{n+1}^+ + S_{n+1}^-) - \frac{1}{4}(S_n^+ - S_n^-)(S_{n+1}^+ - S_{n+1}^-) \tag{9}$$

$$= \frac{1}{2}(S_n^+ S_{n+1}^- + S_n^- S_{n+1}^+) \tag{10}$$

So that, the total Hamiltonian becomes:

$$H = -J \sum_{i=1}^{N} \left[\frac{1}{2}(S_n^+ S_{n+1}^- + S_n^- S_{n+1}^+) + S_n^z S_{n+1}^z \right]$$

```
In [58]: def hamiltonianHeisenberg(N = 5, J = 1, periodic = True):
             '''Returns operator corresponding to the Heisenberg 1D spin-chain on N sites.
             ££ H = - J \sum_{i=1}^N S_n \cdot S_{n+1} ££
             where £ S_n (S_x, S_y, S_z) £ is the spin-operator acting on the n^{th} site.

             H can be written in terms of spin-flip operators £S^+, S^-£ as:

             ££ H = -J \sum_{i=1}^N \left[ \frac{1}{2} (S_n^+ S_{n+1}^- + S_n^- S_{n+1}^+) + S_n^z S_{n

             '''

             spinp_list = []
             spinm_list = []
             spinz_list = []

             opSpinP = sigmap()
             opSpinM = sigmam()
             opSpinZ = sigmaz()

             idOp = identity(2)

             idOp_list = []

             [idOp_list.append(idOp) for i in range(N)]
```

```
    superid = tensor(idOp_list) # identity operator for whole system

    H = 0

    for i in range(N):
        # Create list containing creation/destruction/number operators for each site

        spinp_list.append(posOperatorN(opSpinP,N=N,i=i))
        spinm_list.append(posOperatorN(opSpinM,N=N,i=i))
        spinz_list.append(posOperatorN(opSpinZ,N=N,i=i))

    if periodic == True:
        for i in range(N):
            H += - J * ( 0.5*(spinp_list[i%N] * spinp_list[(i+1)%N] + spinm_list[i%N] * spinm_l
                    + spinz_list[i%N] * spinz_list[(i+1)%N] )
    else:
        for i in range(N-1):
            H += - J * ( 0.5*(spinp_list[i] * spinp_list[i+1] + spinm_list[i] * spinm_list[i+1]
                    + spinz_list[i] * spinz_list[i+1] )

    return H
```

In [59]:
```
hamiltonianHeisenberg(N=6)
```

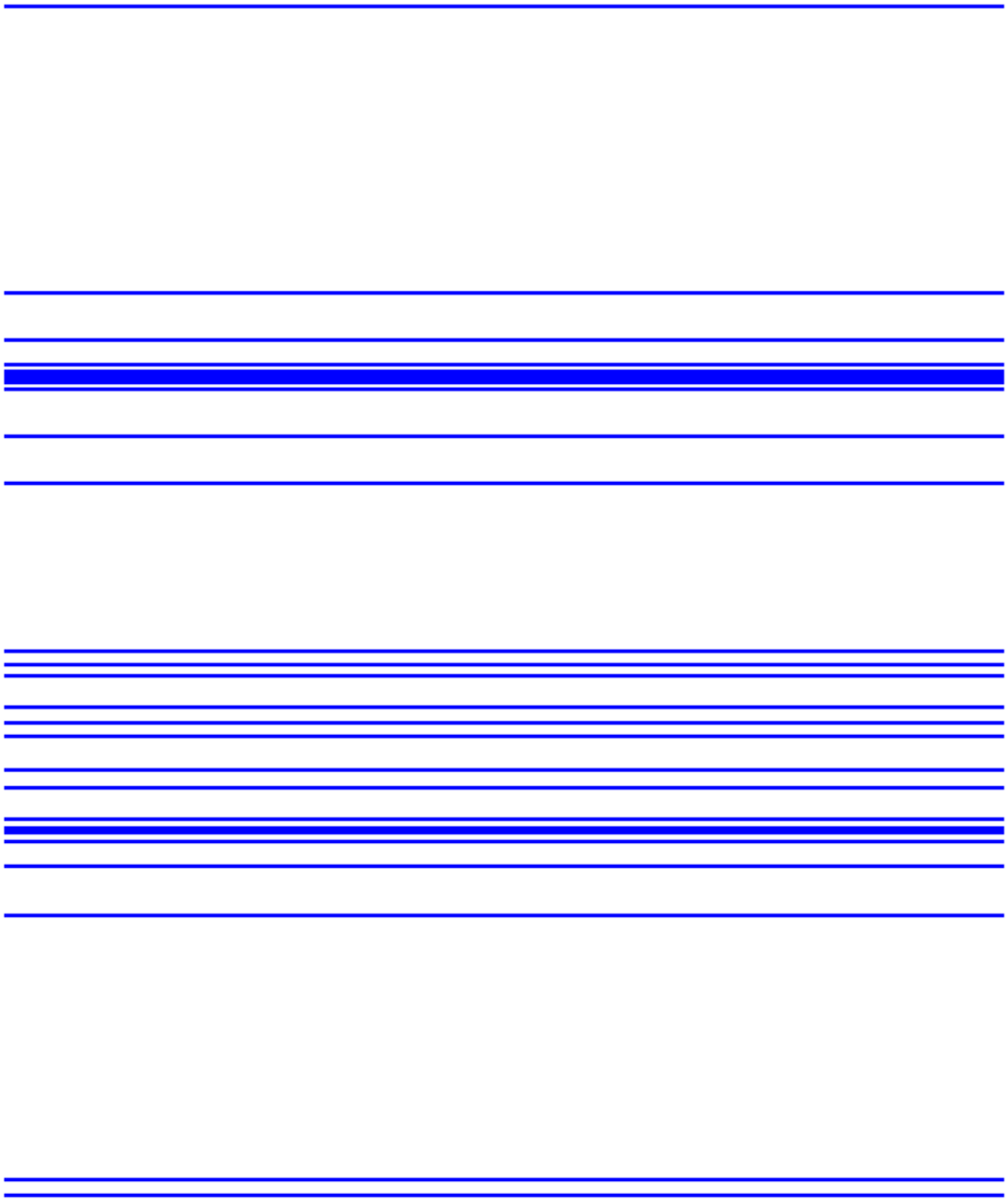
Out[59]:

Quantum object: dims = [[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]], shape = [64, 64], type = oper, isherm = True

$$\begin{pmatrix}
-6.0 & 0.0 & 0.0 & -0.500 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -2.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -2.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.500 & 0.0 & 0.0 & -2.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -2.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -2.0 & 0.0 & 0.0 & -0.500 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -2.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & -2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -0.500 & 0.0 & 0.0 & -6.0
\end{pmatrix}$$

In [60]:
```
plot_energy_levels([_59])
```

Out[60]:
```
(<matplotlib.figure.Figure at 0x109782f10>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109573210>)
```



In [61]:
```
hamiltonianHeisenberg(N=8,periodic=False)
```

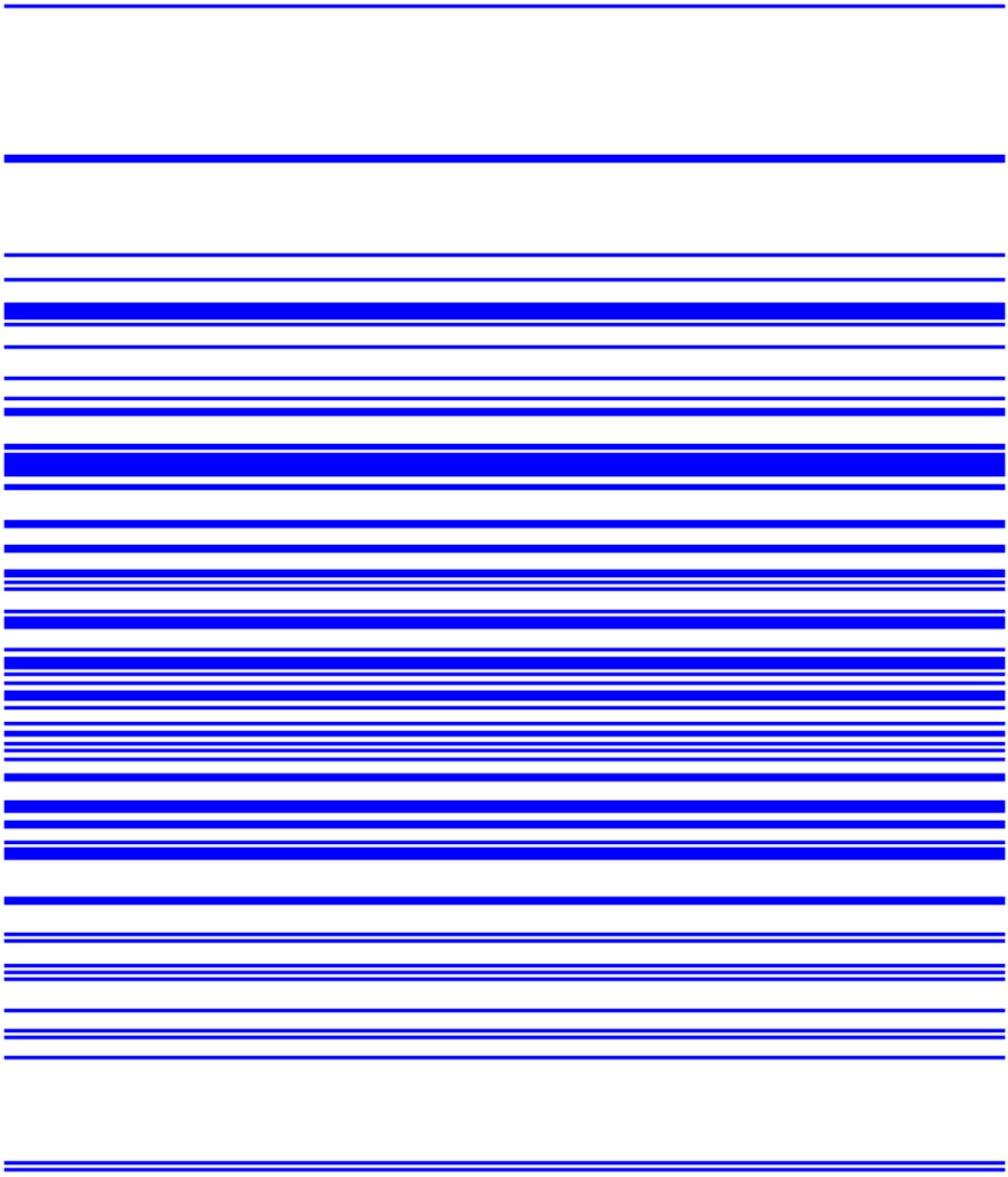
Out[61]:

Quantum object: dims = [[2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2]], shape = [256, 256], type = oper, isherm = True

$$\begin{pmatrix}
-7.0 & 0.0 & 0.0 & -0.500 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -5.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -3.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.500 & 0.0 & 0.0 & -5.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -3.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -3.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -5.0 & 0.0 & 0.0 & -0.500 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -3.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & -5.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -0.500 & 0.0 & 0.0 & -7.0
\end{pmatrix}$$

In [62]:
```
plot_energy_levels([_61])
```

Out[62]:
```
(<matplotlib.figure.Figure at 0x109a87f10>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109a87d90>)
```



In [63]:
```
hamiltonianHeisenberg(N=8,periodic=True)
```

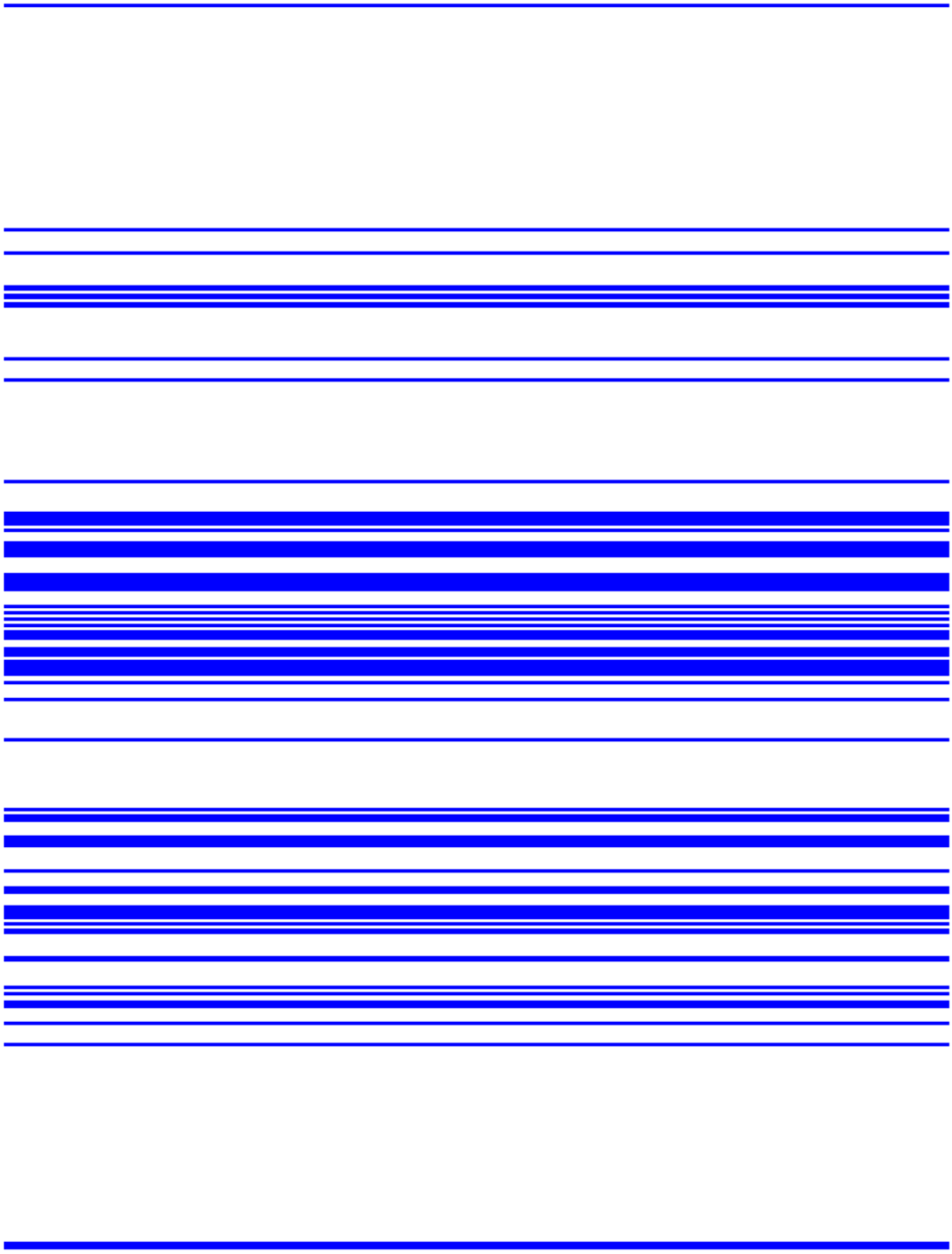
Out[63]:

Quantum object: dims = [[2, 2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2, 2]], shape = [256, 256], type = oper, isherm = True

$$\begin{pmatrix}
-8.0 & 0.0 & 0.0 & -0.500 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -4.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -4.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-0.500 & 0.0 & 0.0 & -4.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -4.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -4.0 & 0.0 & 0.0 & -0.500 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -4.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & -4.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & -0.500 & 0.0 & 0.0 & -8.0
\end{pmatrix}$$

In [64]:
```
plot_energy_levels([_63])
```

Out[64]:
```
(<matplotlib.figure.Figure at 0x10b4f6e90>,
 <matplotlib.axes._subplots.AxesSubplot at 0x10b4e62d0>)
```



0.6 Vacuum State, Basis States

In [65]:
```
def stateVacuum(N = 10):
    ''' Returns a QuTiP object representing the vacuum state for a spin-chain with N sites '''

    state = []

    for i in range(N):
        state.append(basis(2,0))

    return tensor(state)
```

In [66]:
```
stateVacuum(N=3)
```

Out[66]:

Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket

$$\begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

In [67]:
```
def stateUpK(k = 0, N = 10):
    ''' Returns a QuTiP object representating a state with the k^th spin pointing up and the r
    down for a N site spin-chain. '''

    state = []

    for i in range(N):
        state.append(basis(2,0))

    state[k] = basis(2,1)

    return tensor(state)
```

In [68]:
```
stateUpK(2,3)
```

Out[68]:

Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket

$$\begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

In [69]:
```
def stateDownK(k = 0, N = 10):
    ''' Returns a QuTiP object representating a state with the k^th spin pointing down and the
    up for a N site spin-chain. '''

    state = []

    for i in range(N):
        state.append(basis(2,1))

    state[k] = basis(2,0)

    return tensor(state)
```

In [70]:
```
stateDownK(2,3)
```

Out[70]:

Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket

$$\begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \end{pmatrix}$$

In [71]:
```
def stateBasisListDown(N = 10):
    ''' Returns a list of QuTiP objects, whose elements are the mutually orthogonal basis stat
    down spins, of a spin-chain with N sites. '''

    basisList = []

    for i in range(N):
        basisList.append(stateDownK(i,N))

    return basisList

def stateBasisListUp(N = 10):
    ''' Returns a list of QuTiP objects, whose elements are the mutually orthogonal basis stat
    up spins, of a spin-chain with N sites. '''

    basisList = []

    for i in range(N):
        basisList.append(stateUpK(i,N))

    return basisList
```

In [72]:
```
stateBasisListDown(N=3)
```

Out[72]:
```
[Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
 Qobj data =
 [[ 0.]
  [ 0.]
  [ 0.]
```

```
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]],
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]],
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]]
```

0.7 Jordan-Wigner Transformation

Given a n-qubit system, with Pauli operator $ X_i, Y_i, Z_i $ acting on each qubit, we can define a set of **fermionic** operators $ {a_j} $

$$a_j = -\left(\otimes_{i=1}^{j-1} Z_i\right) \otimes \sigma_j$$

The function `jordanWignerDestroyI(i,N)` returns the fermionic destruction operator a_i for the i^{th} site of a N-site qubit chain.

```
In [74]: def jordanWignerDestroyI(i = 0, N = 1):
             ''' Returns the fermionic annihilation operator for the i^th site of a N site spin-chain:
             a_i = Z_1 x Z_2 ... x Z_{i-1} x sigma_i
             where Z_i is the Pauli sigma_z matrix acting on the i^th site and sigma_i is the density
             matrix acting on the i^th qubit, given by:
             sigma_i = id_1 x id_2 ... x id_{i-1} x |0><1| x id_{i+1} ... x id_n
             where id_i is the identity operator

             Reference: Nielsen, Fermionic CCR and Jordan Wigner Transformation
             '''

             # create zop, assign to it identity operator for N site system
             zop = tensor([qeye(2)]*N)

             # for k in (0..i-1), create Z_k operator for N-site chain
             # zop is product of Z_1 * Z_2 ... * Z_{i-1}

             for k in range(i):
                 zop *= posOperatorN(sigmaz(),i = k, N = N)
```

```
        # create single qubit density matrix |0><1|

        sigmai = ket([0])*bra([1])

        # create list of N single-site identity operators

        op_list = identityList(N = N)

        # assign single qubit density matrix |0><1| to i^th item of above list

        op_list[i] = sigmai

        # take the tensor product of resulting list to obtain operator for density matrix |0><1|
        # acting on i^th qubit of N-site chain.

        sigmaop = tensor(op_list)

        # return zop*sigmaop = Z_1 x Z_2 .. x Z_{i-1} x sigma_i, which is fermionic annihilation
        # operator for i^th site of N-site spin-chain.

        return zop * sigmaop
```

In [75]:
```
jordanWignerDestroyI(i=3,N=7)
```

Out[75]:

Quantum object: dims = [[2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2]], shape = [128, 128], type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

In [76]:
```
a4of7 = jordanWignerDestroyI(i=3,N=7)
```

In [77]:
```
commutator(a4of7, a4of7.dag(), kind='anti')
```

Out[77]:

Quantum object: dims = [[2, 2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2, 2]], shape = [128, 128], type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

In [78]:
```
def offDiagBelow(N,a=1):
    """
    Returns a NxN array with the elements below the diagonal set to a
    """
    return a*(np.tri(N,N,-1) - np.tri(N,N,-2))

def offDiagAbove(N,a=1):
    """
    Returns a NxN array with the elements above the diagonal set to a
    """
    return a*(np.tri(N,N,1) - np.tri(N,N))
```

In [69]:
```
import sympy as sp
```

In [71]:
```
vara, varb = sp.var('a'), sp.var('b')
vara, varb
```

Out[71]:
```
(a, b)
```

In [72]:
```
# sympy can take a numpy object as input and return a sympy object,
# on which we can then perform symbolic computations
sp.Matrix(offDiagAbove(4))
```

Out[72]:
```
Matrix([
[0.0, 1.0, 0.0, 0.0],
[0.0, 0.0, 1.0, 0.0],
[0.0, 0.0, 0.0, 1.0],
[0.0, 0.0, 0.0, 0.0]])
```

In [73]:
```
mat1 = varb*sp.Matrix(offDiagAbove(4)+offDiagBelow(4))
mat1
```

Out[73]:
```
Matrix([
[    0, 1.0*b,     0,     0],
[1.0*b,     0, 1.0*b,     0],
[    0, 1.0*b,     0, 1.0*b],
[    0,     0, 1.0*b,     0]])
```

In [75]:
```
mat1 += vara*sp.eye(4)
mat1
```

Out[75]:
```
Matrix([
[    a, 1.0*b,     0,     0],
[1.0*b,     a, 1.0*b,     0],
[    0, 1.0*b,     a, 1.0*b],
[    0,     0, 1.0*b,     a]])
```

```
In [77]: mat1.eigenvals()

Out[77]: {a + b/2 - sqrt(5)*sqrt(b**2)/2: 1,
          a + b/2 + sqrt(5)*sqrt(b**2)/2: 1,
          a - b/2 + sqrt(5)*sqrt(b**2)/2: 1,
          a - b/2 - sqrt(5)*sqrt(b**2)/2: 1}

In [83]: for k in _77.keys():
             print sp.expand(k)

a + b/2 - sqrt(5)*sqrt(b**2)/2
a + b/2 + sqrt(5)*sqrt(b**2)/2
a - b/2 + sqrt(5)*sqrt(b**2)/2
a - b/2 - sqrt(5)*sqrt(b**2)/2

In [84]: sp.simplify(sp.sqrt(varb**2))

Out[84]: sqrt(b**2)

In [22]: ising.eigenenergies()

Out[22]: array([-2.5, -2.5, -2. , ...,  2. ,  2.5,  2.5])

In [26]: class Hamiltonian(Qobj):

             _hamTypes = ['NumberOp', 'Ising', 'Heisenberg']
             _hamType = ''
             _maxSites = 100
             _numSites = 1
             _dims = 2
             _label = None
             _data = None

             _hamiltonian = Qobj()
             _eigenenergies = []
             _eigenstates = []
             _isHermitian = True

             def __init__(self, label=None, dims=2, isHermitian=True,\
                          numSites=1, hamType=None,data=None):
         #        try:
         #            from qutip import *
         #        except:
         #            raise NameError('QuTiP is not installed')

                 if numSites<1 or not isinstance(numSites,int):
                     raise ValueError('numSites must be an integer greater than or equal to 1')
                 if numSites>self._maxSites:
                     raise ValueError('numSites cannot be greater than ' + str(self._maxSites))
                 else:
                     self._numSites = numSites

                 if label!=None and isinstance(label, str):
                     self._label = label

                 if data!=None:
```

```
            self._data = data

        self._isHermitian = isHermitian

        if hamType != None:
            if hamType not in self._hamTypes:
                from string import join
                raise ValueError('hamType must be one of ' + join(self._hamTypes,', '))
            else:
                self._hamType = hamType
                self.createHamiltonian()
        else:
            self._hamiltonian = Qobj()

        if dims < 2 or not isinstance(dims, int):
            raise ValueError('dim must be an integer greater than or equal to 2')
        else:
            self._dims = dims

        Qobj.__init__(self._hamiltonian)

        return

    def createHamiltonian(self):

        if self._hamType == 'Ising':

            self._hamiltonian = isingHamiltonian(self._numSites,self._data['jcoefs'],self._data

        elif self._hamType == 'Hubbard':

            self._hamiltonian = hubbardHamiltonian(self._numSites, self._data['t'],\
                                self._data['U'], self._data['mu'], \
                                self._data['shift'], self._dims)

        elif self._hamType == 'NumberOp':

            numOp = create(self._dims)*destroy(self._dims)

            self._hamiltonian = operHamiltonian(numOp, self._numSites, \
                                    self._data['coefs'], self._dims)
        return

    @property
    def hermitian(self):
        return self._isHermitian

    @hermitian.setter
    def hermitian(self, value):
        if isinstance(value, bool):
            self._isHermitian = value
        else:
            raise ValueError('hermitian must be a boolean data type')
```

```
In [57]: h2 = Hamiltonian()

In [50]: h2._hamTypes

Out[50]: ['NumberOp', 'Ising', 'Heisenberg']

In [58]: h2.eigenenergies()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-58-6533ad75d2a1> in <module>()
----> 1 h2.eigenenergies()


/Users/deepak/anaconda/lib/python2.7/site-packages/qutip/qobj.pyc in eigenenergies(self, sparse
   1321
   1322         """
-> 1323         return sp_eigs(self.data, self.isherm, vecs=False, sparse=sparse,
   1324                        sort=sort, eigvals=eigvals, tol=tol, maxiter=maxiter)
   1325


AttributeError: 'Hamiltonian' object has no attribute 'data'
```

```
In [37]: np.linalg.svd?
```

0.8 Schmidt Decomposition (Singular Value Decomposition)

We need to be able to perform the following tests:

1. Determine whether a given state is entangled or not
2. Determine whether a given density matrix represents a pure or a mixed state

In order to test of entanglement, we can use the Schmidt decomposition or the Singular Value Decomposition (SVD). Numpy implements the SVD in the module numpy.linalg.svd, which given a matrix A as argument, returns the SVD of A, in the form of two unitary matrices U, V and a diagonal matrix S:

$$A = U \cdot S \cdot V$$

Since S is diagonal ($ S_{ij} \equiv s_i \delta_{ij}$), $the elements of A$ can be written as:

$$\begin{aligned} A_{ij} &= U_{ik} \cdot S_{km} \cdot V_{mj} && (11)\\ &= U_{ik} \cdot s_k \delta_{km} \cdot V_{mj} && (12)\\ &= s_k U_{ik} V_{kj} && (13) \end{aligned}$$

Construct entangled state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

The state $|\psi\rangle$ can be written in the form:

$$\psi = \alpha_{ij} u_i \otimes v_j$$

where $u_i, v_i \in \{|0\rangle, |1\rangle\}$ are basis vectors for the two Hilbert spaces H_1 and H_2. For the given state, the matrix α_{ij} has the form:

$$\alpha = \begin{pmatrix} 0.707 & 0 \\ 0 & 0.707 \end{pmatrix}$$

If $|\psi\rangle$ is **not** entangled then in the SVD of the matrix α_{ij}:

$$\alpha = U\, S\, V$$

the diagonal matrix S will have only one non-zero element.

```
In [41]: alpha = np.array([[0.707,0],[0,0.707]])
         alpha

Out[41]: array([[ 0.707,  0.   ],
                [ 0.   ,  0.707]])

In [42]: alpha_svd = np.linalg.svd(alpha)
         alpha_svd

Out[42]: (array([[ 1.,  0.],
                 [ 0.,  1.]]), array([ 0.707,  0.707]), array([[ 1.,  0.],
                 [ 0.,  1.]]))
```

From which we see that: $U = V = \mathbf{1}$ and \$ S = diag(0.707,0.707) \$, where S has more than one non-zero element and therefore we conclude that $|\psi\rangle$ is entangled.

Of course, this is a trivial example, since the SVD can be read off from looking the expression for α_{ij}.

0.9 Sandbox

```
In [123]: hamiltonianIsing(N = 4, jcoefs = [-1,-1,-1,-1])

Out[123]:
```

Quantum object: dims = [[2, 2, 2, 2], [2, 2, 2, 2]], shape = [16, 16], type = oper, isherm = True

$$\begin{pmatrix} -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 \end{pmatrix}$$

```
In [125]: _123.eigenenergies()

Out[125]: array([-1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                  0.,  1.,  1.])

In [104]: hamiltonianIsing(N = 3)
```

Out[104]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True

$$\begin{pmatrix} -0.500 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.500 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.500 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.500 \end{pmatrix}$$

```
In [ ]: hami
```

```
In [73]: psi.dims?
```

```
In [77]: [basis(2,0)]*3
```

```
Out[77]: [Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
         Qobj data =
         [[ 1.]
          [ 0.]], Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
         Qobj data =
         [[ 1.]
          [ 0.]], Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
         Qobj data =
         [[ 1.]
          [ 0.]]]
```

Construct state corresponding to:

```
In [85]: psi2 = 1/(np.sqrt(2)) * (tensor([basis(2,0)]*3) + tensor([basis(2,1)]*3))
         psi2
```

Out[85]:

Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket

$$\begin{pmatrix} 0.707 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.707 \end{pmatrix}$$

```
In [94]: psi2.dims
```

```
Out[94]: [[2, 2, 2], [1, 1, 1]]
```

```
In [98]: num(3).unit()
```

Out[98]:

Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.333 & 0.0 \\ 0.0 & 0.0 & 0.667 \end{pmatrix}$$

In [102]:
```
basis(3,1)
```

Out[102]:

Quantum object: dims = [[3], [1]], shape = [3, 1], type = ket

$$\begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \end{pmatrix}$$

In [113]:
```
aj = sigmaz()*ket([0])*bra([1])
aj
```

Out[113]:

Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 1.0 \\ 0.0 & 0.0 \end{pmatrix}$$

In [116]:
```
commutator(aj,aj.dag(),kind='anti')
```

Out[116]:

Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

In [144]:
```
sz = sigmaz()
```

In [156]:
```
sz1 = tensor(sz,qeye(2),qeye(2))
sz2 = tensor(qeye(2),sz,qeye(2))
sz1, sz2
```

Out[156]:
```
(Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True
 Qobj data =
 [[ 1.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  1.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  1.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  1.  0.  0.  0.  0.]
  [ 0.  0.  0.  0. -1.  0.  0.  0.]
  [ 0.  0.  0.  0.  0. -1.  0.  0.]
  [ 0.  0.  0.  0.  0.  0. -1.  0.]
  [ 0.  0.  0.  0.  0.  0.  0. -1.]],
 Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True
 Qobj data =
 [[ 1.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  1.  0.  0.  0.  0.  0.  0.]
  [ 0.  0. -1.  0.  0.  0.  0.  0.]
  [ 0.  0.  0. -1.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  1.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  1.  0.  0.]
  [ 0.  0.  0.  0.  0.  0. -1.  0.]
  [ 0.  0.  0.  0.  0.  0.  0. -1.]])
```

In [157]:
```
sigma1 = tensor(qeye(2),qeye(2),(ket([0])*bra([1])))
sigma1
```

Out[157]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

In [158]:
```
aj1 = sz1 * sz2 * sigma1
aj1
```

Out[158]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

In [165]:
```
commutator(aj1,aj1.dag(),kind='anti') - tensor([qeye(2)]*3)
```

Out[165]:

Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

In [129]:
```
M = tensor([sigmaz()]*4)
M
```

Out[129]:

Quantum object: dims = [[2, 2, 2, 2], [2, 2, 2, 2]], shape = [16, 16], type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

```
In [130]: M.diag()

Out[130]: array([ 1., -1., -1.,  1., -1.,  1.,  1., -1., -1.,  1.,  1., -1.,  1.,
                 -1., -1.,  1.])
```

Construct density matrix corresponding to given state:
%

$$|\psi\rangle\langle\psi| =$$

```
In [31]: ground = h1.eigenstates()

In [45]: tri = np.tri(3)
         Qobj(tri)

Out[45]:
```

Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isherm = False

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{pmatrix}$$

```
In [63]: offDiagAbove(4)

Out[63]: array([[ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.],
                [ 0.,  0.,  0.,  0.]])

In [68]: Qobj(offDiagBelow(5) + offDiagAbove(5))

Out[68]:
```

Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \end{pmatrix}$$

```
In [60]: np.tri(4,4,1) - np.tri(4,4)

Out[60]: array([[ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.],
                [ 0.,  0.,  0.,  0.]])

In [62]: np.tri??

In [49]: np.triu_indices?

In [32]: np.shape(ground)

Out[32]: (2, 64)

In [33]: Qobj.expm?

In [34]: (2j*np.pi*sigmaz()).expm()
```

Out[34]:

Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

```
In [35]: exph1 = (-0.95*h1).expm()

In [36]: exph1.tr()

Out[36]: 1.5002946744691417

In [37]: (h1*exph1).tr()

Out[37]: 1.6477820449445226

In [38]: (exph1 * h1).tr()/exph1.tr()

Out[38]: 1.0983056015496204

In [39]: expect?

In [40]: expect(h1,ground[1][10])

Out[40]: 5.6

In [41]: entropy_vn(h1)

Out[41]: -1194.5064227355053

In [16]: basis(4,0)
```

Out[16]:

Quantum object: dims = [[4], [1]], shape = [4, 1], type = ket

$$\begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$