ex04_solution

March 15, 2025

Computational Quantum Physics - PHYS 463

Lecturer: Prof. G. Carleo

Assistants: alessandro.sinibaldi@epfl.ch, linda.mauron@epfl.ch, lorenzo.fioroni@epfl.ch

0.1 Solution 04: Many-body systems of indistinguishable particles

[1]:
```
# common imports

# scipy consists of a lot of different modules which we have to import␣
 ↪separately
import scipy
import scipy.sparse
import scipy.sparse.linalg

# use csr_array rather than csr_matrix to have the same operators as with␣
 ↪numpy, i.e.
# @ is the matrix product and * is the elementwise product for csr_array and np.
 ↪array
# csr_matrix uses both @ and * for the matrix product
from scipy.sparse import csr_array

import matplotlib.pyplot as plt

import numpy as np

# nicer printing of floating point numbers
np.set_printoptions(precision=3,suppress=1e-8,linewidth=120)
```

[2]:
```
import functools # for functools.reduce
```

throughout this exercise we have to compute a lot of expressions of the form

```
res = kron(x0, kron(x1, kron(x2, kron(x3, ...))))
```

the pedestrian way of achieving this would be to use a loop:

```
res = 1
for xi in [..., x3, x2, x1, x0]:
    res = kron(x, res)
```

however we can do the same in a functional way in just 1 line (using `functools.reduce`):

```
res = reduce(kron, [x0, x1, x2, x3, ...])
```

0.1.1 Problem 4.2 - Exact Diagonalization of 2-site Lattice Hamiltonians

We are interested in finding static properties of quantum systems, moving from spin systems in the previous exercise to systems of indistiguishable particles.

bosons Given how the creation and annihilation operators act on the bosonic Fock space, we choose the second quantization formalism and create the matrices accordingly.

We have to keep in mind that we need to truncate the Hilbert space when we are considering bosons.

Given a finite cutoff d write the destruction operator

$$\hat{b} = \begin{pmatrix} 0 & \sqrt{1} & 0 & \cdots & 0 \\ 0 & 0 & \sqrt{2} & \cdots & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \sqrt{d-1} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\hat{b}_i = \underbrace{\hat{I}(d) \otimes \ldots \hat{I}(d)}_{i-1 \text{ terms}} \otimes \hat{b} \otimes \underbrace{\hat{I}(d) \otimes \ldots \hat{I}(d)}_{L-i \text{ terms}},$$

[3]:
```
def b(i, d, L):
    '''
    bosonic annihilation operator

    Args:
        i: state to act on
        d: occupation number cutoff (not inclusive)
        L: number of states

    Returns:
        A sparse matrix representing the destruction of a boson in state i
    '''
    b_    = scipy.sparse.diags(np.sqrt(np.arange(1, d)), 1)

    left  = scipy.sparse.identity(d ** i)
    right = scipy.sparse.identity(d ** (L - i - 1))
    mat   = functools.reduce(scipy.sparse.kron, (left, b_, right))
    return mat
```

For the creation operator $b^\dagger$, we only need to transpose b (and take the conjugate, but since this is a real operator this operation is superfluous) .

Using this write a function for the number operator acting on state i

$$\hat{n}_i = \hat{b}_i^\dagger b_j$$

[4]:
```
def n_b(i,d,L):
    '''
    bosonic number operator

    Args:
        i: index of state to act on
        d: occupation number cutoff (not inclusive)
        L: number of states

    Returns:
        A sparse matrix counting the number of bosons in state i
    '''

    bi     = b(i,d,L)

    bi_dagger = bi.conjugate().T
    ni = bi_dagger @ bi

    return ni
```

and the total number operator

$$\hat{N} = \sum_i \hat{n}_i$$

[5]:
```
def N_b(d,L):
    '''
    bosonic total number operator

    Args:
        d: occupation number cutoff (not inclusive)
        L: number of states

    Returns:
        A sparse matrix which counts the total number of bosons
    '''
    return sum(n_b(i,d,L) for i in range(L))
```

Now define a function to create the Bose-Hubbard Hamiltonian for constant t and U using the operators defined above.

$$\hat{H} = -t \sum_{\langle i,j \rangle} \left(\hat{b}_i^\dagger \hat{b}_j + \hat{b}_j^\dagger \hat{b}_i \right) + \frac{U}{2} \sum_i \hat{n}_i \left(\hat{n}_i - 1 \right) - \mu \sum_i \hat{n}_i$$

[6]:
```
def bose_hubbard(d=5, L=2, t=1, U=1, mu=0, pbc=False):
    """
    bose-hubbard hamiltonian

    Args:
        d: occupation number cutoff (not inclusive)
        L: number of states
        t: hopping amplitude
        U: interaction term
        pbc: whether to use periodic boundary conditions (True) or
             open boundary conditions (False, default)
    """
    if pbc and L > 2:
        edges = [(i, (i+1)%L) for i in range(L)]
    else:
        edges = [(i, i+1) for i in range(L-1)]

    hopping   = 0
    for i,j in edges:
        bi = b(i, d, L)
        bj = b(j, d, L)
        hopping   += bi.T.conj() @ bj + bj.T.conj() @ bi

    local_int = 0
    for i in range(L):
        n_i  = n_b(i, d, L)
        local_int += n_i @ n_i - n_i

    H = -t * hopping + 0.5 * U * local_int

    # explicitly convert to csr
    H = csr_array(H)
    return H
```

At this point we have all the ingredients to create the Hamiltonian for our problem and diagonalize it

- $U = 1$ and $t = 1$

[7]:
```
L = 2
d = 5

U = 1
t = 1
```

[8]:
```
H = bose_hubbard(d, L, t=t, U=U)
N = N_b(d,L)
```

Diagonalize the hamiltonian using scipy:

[9]:
```
eigvals, eigvecs = scipy.sparse.linalg.eigsh(H, which='SA')
eigvals
```

[9]:
```
array([-1.646, -1.562, -1.247, -1.   , -0.275,  1.   ])
```

Optionally you can verify that your results match those those computed using our library NetKet (https://github.com/netket/netket)

[10]:
```
#check with netket

import netket as nk
g = nk.graph.Chain(L, pbc=False)
hi = nk.hilbert.Fock(n_max=d-1, N=L)
ha = nk.operator.BoseHubbard(hi, g, U=U, J=t)
nk.exact.lanczos_ed(ha, k=6)
```

```
/home/fioroni/Documents/Research/2025/TA_computational_quantum_physics/.venv/lib
/python3.12/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found.
Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
WARNING:2025-03-15 12:58:12,639:jax._src.xla_bridge:966: An NVIDIA GPU may be
present on this machine, but a CUDA-enabled jaxlib is not installed. Falling
back to cpu.
```

[10]:
```
array([-1.646, -1.562, -1.247, -1.   , -0.275,  1.   ])
```

We know that the hamiltonian $\hat{H}$ commutes with the total number operator $\hat{N}$, therefore all eigenstates $|\psi_i\rangle$ of $\hat{H}$ have a fixed number of particles.

Compute the number of particles of each eigenstate by taking the expectation value of the total number operator $\langle\psi_i|\hat{N}|\psi_i\rangle$:

[11]:
```
print(' i\t   Ei\t  N  \n')
for i, psi in enumerate(eigvecs.T):
    n = psi.T.conj() @ N @psi
    print(f'{i}\t {f"{eigvals[i]:0.3f}".rjust(6)}\t {n:0.1f}')
```

```
 i         Ei     N

0        -1.646  3.0
1        -1.562  2.0
2        -1.247  4.0
3        -1.000  1.0
4        -0.275  5.0
5         1.000  1.0
```

Of course, no site is occupied by more than the cut-off d

[12]:
```
nops = [n_b(j,d,L) for j in range(L)]
ns = np.zeros(L)

print(' i\t   Ei\t', ''.join(map(str, [f' n_{i}  ' for i in range(L)])), '\t␣
 ↪n_k  < d \n')
for i, psi in enumerate(eigvecs.T):
    for j in range(L):
        ns[j] = (psi.T.conj() @ nops[j] @psi)
    print(f'{i}\t {f"{eigvals[i]:0.3f}".rjust(6)}\t {ns}\t {np.round(ns)<d}')
```

```
i         Ei     n_0  n_1      n_k  < d

0        -1.646  [1.5 1.5]       [ True  True]
1        -1.562  [1. 1.]         [ True  True]
2        -1.247  [2. 2.]         [ True  True]
3        -1.000  [0.5 0.5]       [ True  True]
4        -0.275  [2.5 2.5]       [ True  True]
5         1.000  [0.513 0.513]   [ True  True]
```

Repeat the same for

- $U = 4$ and $t = 1$

[13]:
```
L = 2
d = 5

H = bose_hubbard(d, L, t=1, U=4)
N = N_b(d,L)
eigvals, eigvecs = scipy.sparse.linalg.eigsh(H, which='SA', k=6)
print(' i\t   Ei\t  N \t N  < d \n')
for i, psi in enumerate(eigvecs.T):
    n = psi.T.conj() @ N @psi
    print(f'{i}\t {f"{eigvals[i]:0.3f}".rjust(6)}\t {n:0.1f}\t {np.round(n)<d}')
```

```
i         Ei     N      N  < d

0        -1.000  1.0      True
1        -0.828  2.0      True
2         1.000  1.0      True
3         1.708  3.0      True
4         4.000  2.0      True
5         4.828  2.0      True
```

- $U = -1$ and $t = 1$

[14]:
```
L = 2
d = 5

H = bose_hubbard(d, L, t=1, U=-1)
```

```
N = N_b(d,L)
eigvals, eigvecs = scipy.sparse.linalg.eigsh(H, which='SA', k=6)
print(' i\t   Ei\t\t  N \t N  < d \n')
for i, psi in enumerate(eigvecs.T):
    n = psi.T.conj() @ N @psi
    print(f'{i}\t {f"{eigvals[i]:0.3f}".rjust(8)}\t {n:0.1f}\t {np.round(n)<d}')
```

```
i         Ei             N      N  < d

0         -13.000        7.0     False
1         -12.000        8.0     False
2         -11.424        6.0     False
3          -9.372        5.0     False
4          -7.615        4.0     True
5          -7.275        5.0     False
```

Fixed numbers of particles

Find the ground state for fixed numbers of particles using the power method with an initial guess with the correct number of particles.

[15]:
```
# slightly adapted power method from the previous exercise
# (added support for matrices which don't have shapes powers of 2)

# lambda is a reserved keyword for lambda functions,
# so we can't give the variable like that name....
def power_method(H, n_iters, lambd=1, u0=None):
    if u0 is None:
        # take a random vector as in initial guess
        # TODO this is a real vector but depending on the hamiltonian
        # it might need to be complex
        u = np.random.normal(size=H.shape[0])
    else:
        u = u0

    # construct the propagator
    I = scipy.sparse.identity(len(u))
    prop = lambd * I - H

    old = np.inf
    energies = []
    for i in range(n_iters):
        u = prop @ u

        # normalize the vector
        # this is necessary since the propagator is not unitary
        # and if we do a lot of iterations the floating point numbers in v ␣
 ↪would grow
```

```
        # very quickly to the point where the exponent overflows
        # also it is convenient for computing the expectation value below
        u = u / np.linalg.norm(u)

        # compute the expectation value
        # For a CSR array `(v @ h) @ v` is slow, therefore we do it the other␣
↪way round
        e = u.T.conj() @ (H @ u)

        # stop early if the energy does not improve anymore
        if np.abs((old - e) / e) < 1e-15:
            break

        energies.append(e)
        old = e

    return u, energies
```

Create a (classical) state with the desired number of particles to be used as an initial guess

$$|n_1, n_2 \ldots, n_L\rangle = |n_1\rangle \otimes |n_2\rangle \otimes \ldots \otimes |n_L\rangle$$

[16]:
```
def bosonic_site_state(d, n):
    """
    Args:
        d: occupation number cutoff
        n: number of bosons
    """

    state = np.zeros(d)
    state[n] = 1
    return state

def bosonic_state(d, occupations):
    """
    Args:
        d: occupation number cutoff
        occupations: a list of the number of bosons in each state
    """
    return functools.reduce(np.kron, (bosonic_site_state(d, n) for n in␣
↪occupations))
```

[17]:
```
L = 2
d = 5
t = 1
U = 1
```

[18]:
```
H = bose_hubbard(d, L, t=t, U=U)
```

Manually create the following initial state with 4 particles:

$$|2,2\rangle = |2\rangle \otimes |2\rangle$$

[19]:
```
# n_particles  = 4
n_list = [2,2]

init_state = bosonic_state(d,n_list)
init_state
```

[19]:
```
array([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.])
```

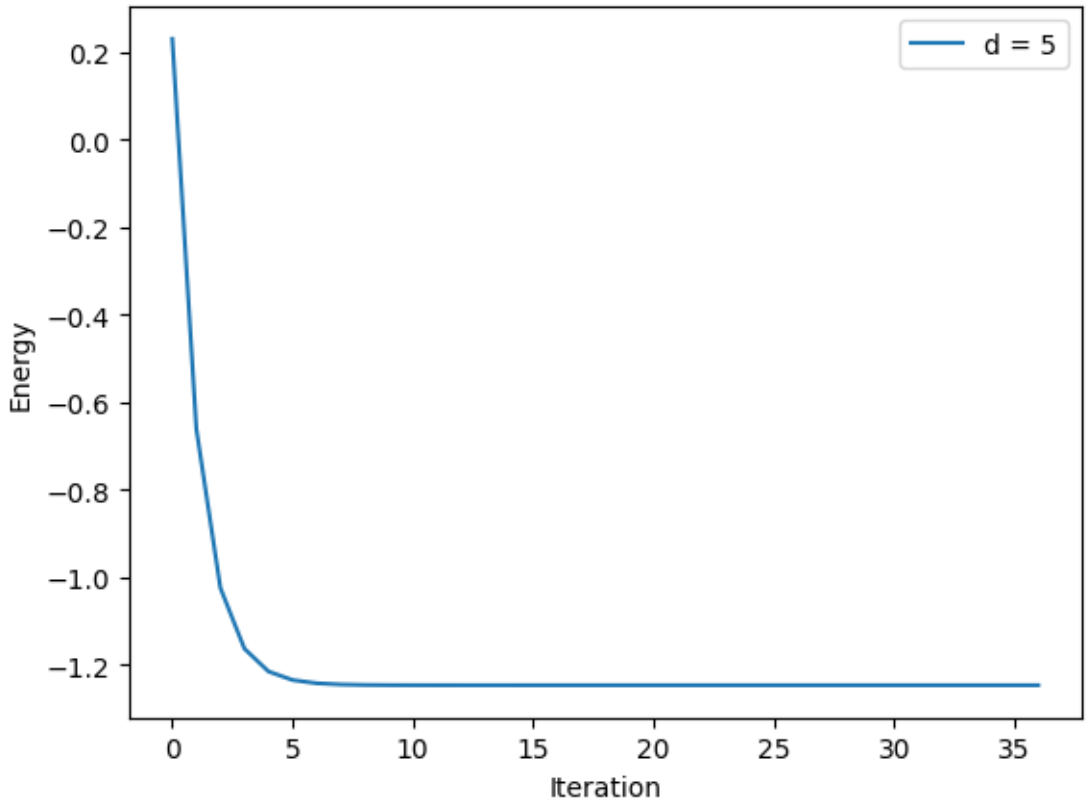
run the power method

[20]:
```
gs, e_gs = power_method(H, 2000, 14., u0=init_state)
e_gs[-1]
```

[20]:
```
np.float64(-1.247303334275856)
```

[21]:
```
plt.plot(np.arange(len(e_gs)),e_gs,label=f"d = {d}")
plt.xlabel("Iteration")
plt.ylabel("Energy")
plt.legend()

plt.show()
```



[22]:
```
#check with netket

import netket as nk
g = nk.graph.Chain(L, pbc=False)
hi = nk.hilbert.Fock(n_max=d-1, N=L, n_particles=4)
ha = nk.operator.BoseHubbard(hi, g, U=U, J=t)
nk.exact.lanczos_ed(ha)
```

[22]:
```
array([-1.247])
```

0.1.2 Problem 4.3 - Exact Diagonalization of the t-V model

spinless fermions Start by defining the pauli matrices $\hat{\sigma}^x, \hat{\sigma}^y, \hat{\sigma}^z$ as well as the spin rising/lowering operators

$$\hat{\sigma}^{\pm} = \frac{\hat{\sigma}^x \pm i\hat{\sigma}^y}{2}.$$

[23]:
```
# pauli matrices
sx = scipy.sparse.csr_array([[0.,  1.],[1., 0.]])
sy = scipy.sparse.csr_array([[0., -1.j],[1.j, 0.]])
sz = scipy.sparse.csr_array([[1., 0.],[0., -1.]])
```

```
I = scipy.sparse.identity(2)

sp = (sx + 1j * sy) / 2
sm = (sx - 1j * sy) / 2
# we know that sp and sm are real:
np.testing.assert_array_equal(sp.imag.todense(), 0)
np.testing.assert_array_equal(sm.imag.todense(), 0)
sp = sp.real
sm = sm.real
```

Construct the fermionic annihilation operator in terms of spin operators

$$\hat{c}_i = \underbrace{\hat{\sigma}^z \otimes \hat{\sigma}^z \ldots \hat{\sigma}^z}_{i-1 \text{ terms}} \otimes \hat{\sigma}^+ \otimes \underbrace{\hat{I} \otimes \hat{I} \cdots \otimes \hat{I}}_{L-i \text{ terms}},$$

[24]:
```
def c(i, L):
    """
    fermionic annihilation operator

    Args:
        i: index of state to act on
        L: number of states

    Returns:
        A sparse matrix destroying a fermion in state i
    """
    # simple (but slow) 1-line version:
    # return functools.reduce(scipy.sparse.kron, [sz,]*i + [sp,] + [sz,]*(L - i␣
 ↪- 1))

    # more optimized:
    # we know that sz is diagonal, therefore dense kron much faster:
    sz_diag = np.array([1, -1])
    l = scipy.sparse.diags_array(functools.reduce(np.kron, [sz_diag]*i, np.
 ↪array([1.])))
    r = scipy.sparse.identity(2**(L - i - 1))
    return functools.reduce(scipy.sparse.kron, [l, sp, r])
```

as well as the number operator

[25]:
```
def n_c(i,L):
    """
    fermionic number operator

    Args:
        i: index of state to act on
        L: number of states
```

```
    Returns:
        A sparse matrix counting the number of fermions in state i
    """

    ci      = c(i, L)
    ci_dagger = ci.conjugate().T
    return ci_dagger @ ci
```

[26]:
```
def N_c(L):
    '''
    fermionic total number operator

    Args:
        L: number of states

    Returns:
        A sparse matrix counting the total number of fermions
    '''
    return sum(n_c(i,L) for i in range(L))
```

Define a function for the t-V hamiltonian

$$H = t \sum_{\langle i,j \rangle} \left(\hat{c}_i^\dagger \hat{c}_j + \hat{c}_j^\dagger \hat{c}_i \right) + V \sum_{\langle i,j \rangle} \hat{n}_i \hat{n}_j$$

[27]:
```
def tV(L, t=1, V=1, pbc=False):

    """
    t-V hamiltonian for a 1d chain of spinless fermions
    with nearest-neighbour interaction term

    Args:
        L: number of states
        t: hopping amplitude
        V: interaction term
        pbc: whether to use periodic boundary conditions (True) or
             open boundary conditions (False, default)

    """

    if pbc and L > 2:
        edges = [(i, (i+1)%L) for i in range(L)]
    else:
        edges = [(i, i+1) for i in range(L-1)]

    H = 0
    for i,j in edges:
```

```
        ci = c(i, L)
        cj = c(j, L)
        ni = n_c(i, L) # = ci.T.conj() @ ci
        nj = n_c(j, L) # = cj.T.conj() @ cj
        H = H + t * (ci.T.conj() @ cj + cj.T.conj() @ ci)
        H = H + V * (ni @ nj)

    # explicitly convert to csr
    H = csr_array(H)
    H.eliminate_zeros()
    return H

tV(4).todense().astype(int)
```

[27]:
```
array([[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, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 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, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3]])
```

Open boundary conditions:

[28]:
```
print('L', '\t', "E0")

for L in range(2,17):
    E0 = scipy.sparse.linalg.eigsh(tV(L), which='SA', k=1)[0][0] / L
    print(L, '\t', "{:0.4f}".format(E0))
```

```
L       E0
2       -0.5000
3       -0.4714
4       -0.5000
5       -0.5177
6       -0.5000
7       -0.5262
8       -0.5205
9       -0.5270
```

```
10      -0.5304
11      -0.5258
12      -0.5337
13      -0.5309
14      -0.5342
15      -0.5353
16      -0.5336
```

Periodic boundary conditions:

[29]:
```
# pbc
print('L', '\t', "E0")
for L in range(2,17):
    E0 = scipy.sparse.linalg.eigsh(tV(L, pbc=True), which='SA', k=1)[0][0] / L
    print(L, '\t', "{:0.4f}".format(E0))
```

```
L       E0
2       -0.5000
3       -0.3333
4       -0.5000
5       -0.6000
6       -0.5393
7       -0.5090
8       -0.5617
9       -0.5556
10      -0.5298
11      -0.5484
12      -0.5551
13      -0.5385
14      -0.5418
15      -0.5523
16      -0.5466
```

Fixed number of particles

Construct a initial state with a fixed number of fermions to use as initial guess for the power method.

[30]:
```
n_particles = 4
L = 8

# we distribute the particles with uniform probability
# select N states randomly
indices = np.random.choice(L, n_particles, replace=False)

n_list = np.zeros(L, dtype=int)
```

```
n_list[indices] = 1
print(n_list)

# reuse the code for bosons from above
# with a cutoff of d=2 since there can be at most 1 fermion in each state
u0 = bosonic_state(2, n_list)
```

```
[0 1 1 0 1 0 1 0]
```

[31]:
```
power_method(tV(L), 1000, 10., u0)[1][-1]
```

[31]:
```
np.float64(-3.9999999999999676)
```

[32]:
```
# check with netket
import netket as nk

def tV_nk(hi, t=1, V=1, pbc=False):
    g = nk.graph.Chain(hi.n_orbitals, pbc=pbc)
    ha = 0
    for i,j in g.edges():
        ci = nk.operator.fermion.destroy(hi,i)
        cj = nk.operator.fermion.destroy(hi,j)
        ci_dagger = nk.operator.fermion.create(hi,i)
        cj_dagger = nk.operator.fermion.create(hi,j)
        ni = nk.operator.fermion.number(hi,i)
        nj = nk.operator.fermion.number(hi,j)
        ha = ha + t * (ci_dagger * cj + cj_dagger * ci)
        ha = ha + V * (ni * nj)
    return ha

hi = nk.hilbert.SpinOrbitalFermions(n_orbitals=L, n_fermions=n_particles)
nk.exact.lanczos_ed(tV_nk(hi), k=1)
```

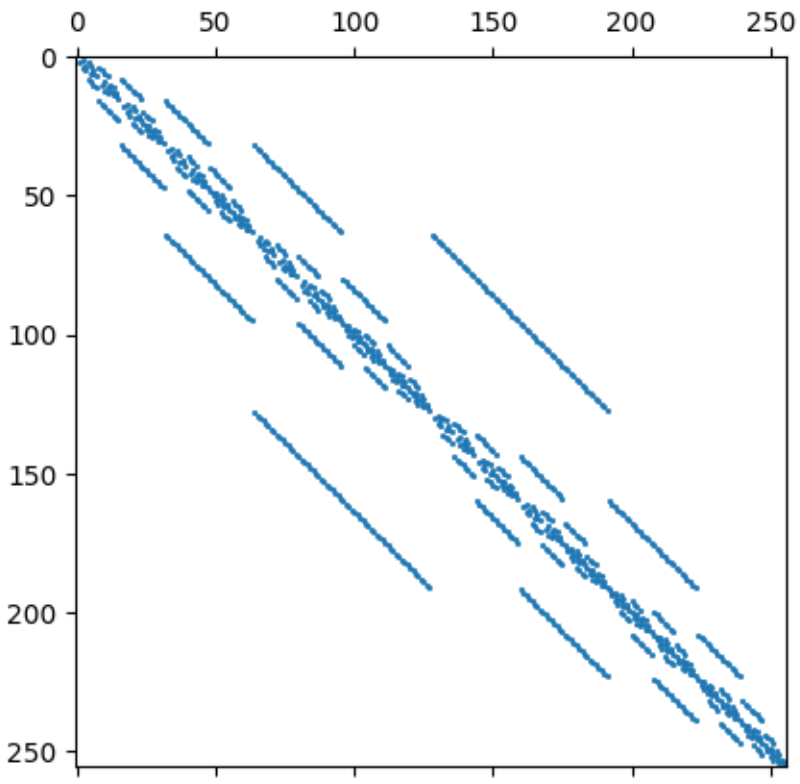
[32]:
```
array([-4.])
```

Block-diagonalize the hamiltonian To do this, find the number of particles of each basis state (which corresponds to the diagonal of $\hat{N}$) and find the suitable permutation to assemble all basis states with same occupation number.

[33]:
```
L = 8

H = tV(L)
plt.spy(H, markersize=1)
```

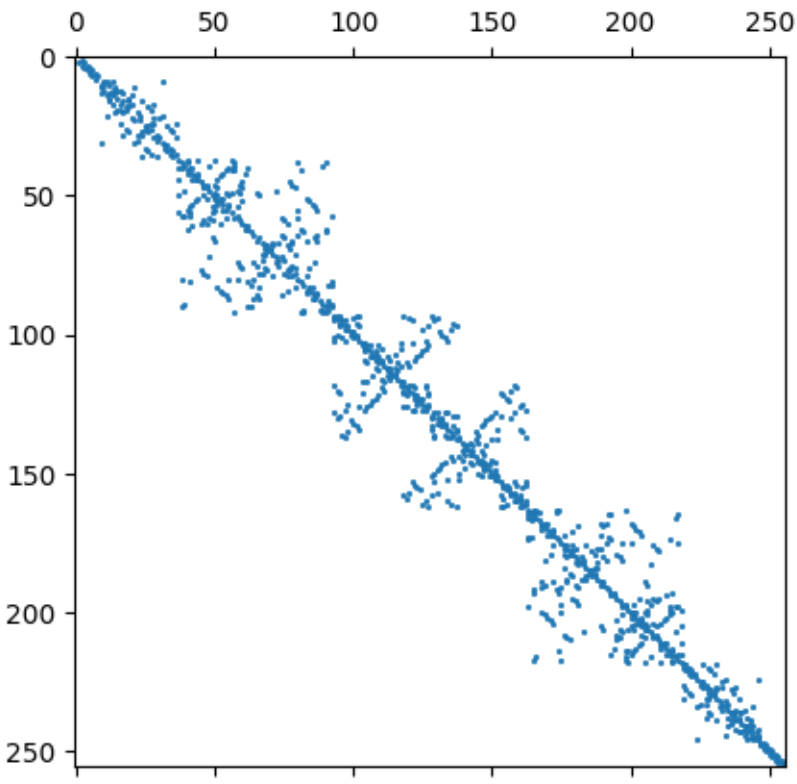
[33]:
```
<matplotlib.lines.Line2D at 0x744620261490>
```



[34]:
```
# we look at the diagonal of the number operator to find the number of fermions
# of each basis state
diag = np.round(N_c(L).diagonal()).astype(int)
# find permutation for increasing number of spins
perm = np.argsort(diag)

# block-diagonalize the matrix
H = H[perm, :][:, perm] # reorder rows and columns
plt.spy(H, markersize=1)
```

[34]: <matplotlib.lines.Line2D at 0x74462b378c20>



[35]:
```
# compute the positions of each block corresponding to a fixed number of particles:
numbers = np.unique(diag)
start = np.searchsorted(diag[perm], numbers, 'left')
end = np.searchsorted(diag[perm], numbers, 'right') # could use np.diff on start w/ the size of the matrix appended
num = end-start

# extract the block corresponding to N=4:
i = int(np.where(numbers == 4)[0][0])
H4 = H[start[i]:end[i], start[i]:end[i]]

# diagonalize the block corresponding to N=4,
# at a greately reduced cost compared to the full Hamiltonian matrix
scipy.sparse.linalg.eigsh(H4, k=2, which='SA')[0]
```

[35]:
```
array([-4.   , -3.164])
```

0.1.3 Problem 4.4 - Exact Diagonalization of the Fermi-Hubbard model

Spinful fermions Basis representation: Number of up and down spins on site 1, number of up and down spins on site 2...

We are dealing with fermions, so each site can hold at most one particle of each spin.

Use the following mapping to spinless fermions:

$$|n_{1\uparrow}, n_{2\uparrow} \dots, n_{L-1,\downarrow}, n_{L,\downarrow}\rangle \rightarrow |n'_1, n'_2 \dots, n'_{2L-1}, n'_{2L}\rangle,$$

$$\hat{c}_{i,\uparrow} \rightarrow \hat{c}'_i \hat{c}_{i,\downarrow} \rightarrow \hat{c}'_{i+L}.$$

(or alternatively like in the lecture notes):

$$|n_{1\uparrow}, n_{1\downarrow} \dots, n_{L,\uparrow}, n_{L,\downarrow}\rangle \rightarrow |n'_1, n'_2 \dots, n'_{2L-1}, n'_{2L}\rangle,$$

$$\hat{c}_{i,\uparrow} \rightarrow \hat{c}'_{2i} \hat{c}_{i,\downarrow} \rightarrow \hat{c}'_{2i+1}.$$

to define the operators in terms of the spinless operators from problem 4.3

[36]:
```python
def c_spin(i, s, L):
    """
    fermionic annihilation operator (with spin)

    Args:
        i: index of state to act on
        s: value of the spin; 0 encodes spin up, and 1 encodes down
        L: number of states

    Returns:
        A sparse matrix destroying a fermion with spin s in state i
    """

    # s: 0 for spin down and 1 for spin up
    return c(i+s*L, 2*L)
    # or alternatively:
    #return c(2*i+s, 2*L)


def n_c_spin(i, s, L):
    """
    fermionic number operator (with spin)

    Args:
        i: index of state to act on
        s: value of the spin; 0 encodes spin up, and 1 encodes down
        L: number of states

    Returns:
        A sparse matrix counting the number of fermions with spin s in state i
```

```
    """

    ci = c_spin(i, s, L)
    return ci.T.conj() @ ci


def N_c_spin(L):
    '''
    fermionic total number operator (with spin)

    Args:
        L: number of states

    Returns:
        A sparse matrix counting the total number of fermions
    '''
    # counting both fermions with spin up and down
    return sum(n_c_spin(i,s,L) for s in [0, 1] for i in range(L))
```

Define the fermi-hubbard hamiltonian (here written for just 2 sites, but you can also implement the general one)

$$\hat{H} = -t \sum_{\sigma=\uparrow,\downarrow} \left(\hat{c}^{\dagger}_{1\sigma} \hat{c}_{2\sigma} + \hat{c}^{\dagger}_{2\sigma} \hat{c}_{1\sigma} \right) + U \sum_{i=1}^{2} \hat{n}_{i\uparrow} \hat{n}_{i\downarrow}.$$

[37]:
```
def fermi_hubbard(L=2, t=1, U=1):
    """
    fermi-hubbard hamiltonian

    Args:
        L: number of states
        t: hopping amplitude
        U: interaction term

    """

    H = 0
    for i,j in [(i, i+1) for i in range(L-1)]:
        for s in [0,1]: # spin down, spin up
            ci = c_spin(i, s, L)
            cj = c_spin(j, s, L)
            H = H - t * (ci.T.conj() @ cj + cj.T.conj() @ ci)

    for i in range(L):
        ni_down = n_c_spin(i, 0, L)
        ni_up = n_c_spin(i, 1, L)
```

```
        H = H + U * ni_up @ ni_down

    # explicitly convert to csr
    H = csr_array(H)
    H.eliminate_zeros()
    return H
```

[38]:
```
H = fermi_hubbard(t=1, U=1)
H.todense().real.astype(int)
```

[38]:
```
array([[ 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, -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,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  1, -1,  0,  0, -1,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0, -1,  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, -1,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0, -1,  0,  0, -1,  1,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, -1,  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,  1, -1,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  1,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2]])
```

and diagonalize it

[39]:
```
scipy.sparse.linalg.eigsh(H, which='SA')[0]
```

[39]:
```
array([-1.562, -1.   , -1.   , -0.   , -0.   , -0.   ])
```

[40]:
```
# check with netket
# this assumes that you chose the first of the two mappings
# if you chose the other one your matrix will look differently

n_particles = 2

hi = nk.hilbert.SpinOrbitalFermions(n_orbitals=n_particles, s=1/2)

def fermi_hubbard_nk(N=2, t=1, U=1, pbc=False):
    g = nk.graph.Chain(hi.n_orbitals, pbc=pbc)
    ha = 0
    for i,j in g.edges():
        for s in [1, -1]: # spin up, spin down
            ci = nk.operator.fermion.destroy(hi, i, sz=s)
            cj = nk.operator.fermion.destroy(hi, j, sz=s)
```

```
            ci_dagger = nk.operator.fermion.create(hi, i, sz=s)
            cj_dagger = nk.operator.fermion.create(hi, j, sz=s)
            ha = ha - t * (ci_dagger*cj + cj_dagger*ci)

    for i in g.nodes():
        ni_up = nk.operator.fermion.number(hi, i, sz=1)
        ni_down = nk.operator.fermion.number(hi, i, sz=-1)
        #
        ha = ha + U * ni_up*ni_down
    return ha

ha = fermi_hubbard_nk(t=1, U=1)
print('H=\n', ha.to_dense().astype(int), '\n')

print('Eigenvalues:\n', nk.exact.lanczos_ed(ha, k=6))
```

```
H=
 [[ 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 -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  0  0  0 -1  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  1 -1  0  0 -1  0  0  0  0  0  0]
 [ 0  0  0  0  0 -1  0  0  0  0 -1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0  0  0 -1  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 -1  0  0  0  0 -1  0  0  0  0  0]
 [ 0  0  0  0  0  0 -1  0  0 -1  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0 -1  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  1 -1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0 -1  1  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2]]

Eigenvalues:
 [-1.562 -1.    -1.    -0.    -0.    -0.   ]
```

In this case, the block-diagonalization can be very efficient, since the Hilbert space is bigger (due to the consideration of the spin degrees of freedom)

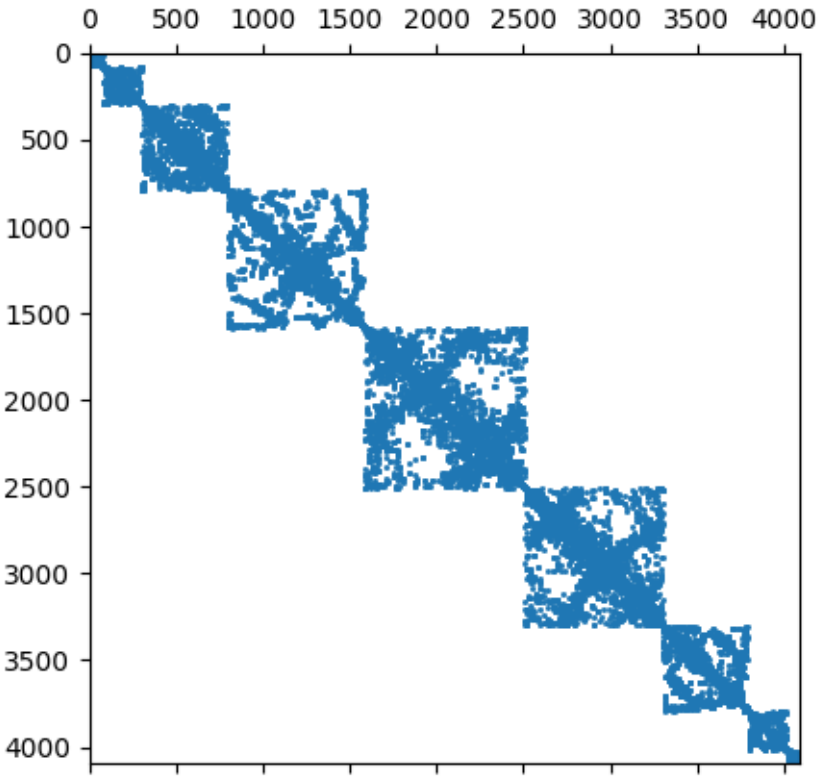
[41]:
```
L = 6

H = fermi_hubbard(L=L, t=1, U=1)

diag = np.round(N_c_spin(L).diagonal()).astype(int)
# find permutation for increasing number of spins
perm = np.argsort(diag)
# block-diagonalize the matrix
```

```
H = H[perm, :][:, perm]
plt.spy(H, markersize=1)
```

[41]: <matplotlib.lines.Line2D at 0x74462210faa0>



[42]:
```
# compute the positions of each block corresponding to a fixed number of␣
 ↪particles:
numbers = np.unique(diag)
start = np.searchsorted(diag[perm], numbers, 'left')
end = np.searchsorted(diag[perm], numbers, 'right')
num = end-start

# extract the block corresponding to N=4:
i = int(np.where(numbers == 4)[0][0])
H4 = H[start[i]:end[i], start[i]:end[i]]

# diagonalize the block corresponding to N=4,
scipy.sparse.linalg.eigsh(H4, k=2, which='SA')[0]
```

[42]: array([-5.478, -4.852])