ex07_solution

April 2, 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 07 - Variational Monte Carlo (VMC)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from functools import partial

try:
    from tqdm.auto import tqdm  # progressbar
except ImportError:
    tqdm = lambda x: x
```

[2]:
```
# analytical solution for the ground state energy of the TFI in 1D with pbc

def ising1d_energy(L, Gamma):
    # Gamma is in units of the interaction strength J
    def Epsilon(k,h):
        eps = 1 + h**2 + 2 * h * np.cos(k)
        return 2 * np.sqrt(eps)

    i = np.arange(L)
    k = np.pi * ( 2 * i + 1)/L
    energy = Epsilon(k,Gamma).sum()
    return -0.5*energy
```

[3]:
```
def err_rel(x, y):
    """relative error"""
    return np.abs((x-y)/y)
```

[4]:
```
N = 16          # number of spins
Ns = 512        # number of samples
N_discard = 128 # how many samples to discard
```

[5]:
```
# throughout this exercise we work with basis states in the  z basis
# represented by arrays containing the quantum numbers -1 and +1

# the following function generates random basis states (uniformly sampled)

def random_states(N, size=1):
    return np.random.choice(np.array([-1,1]), size=(size, N))

random_states(N, 3)
```

[5]:
```
array([[ 1, -1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1],
       [ 1, -1,  1, -1,  1,  1,  1, -1, -1,  1, -1,  1,  1,  1, -1,  1],
       [-1,  1,  1, -1, -1,  1, -1, -1, -1,  1,  1,  1, -1, -1,  1,  1]])
```

0.2 Exercise 7.1 : Mean-field Ansatz

0.2.1 a) Ansatz

[6]:
```
def logpsi_mf(params, s):
    """Mean-field Ansatz"""
    Ns, N = s.shape
    d = params[:N]
    u = params[N:]
    return ((s == -1) * d + (s == +1) * u).sum(axis=-1)

def random_params_mf(N, stddev=0.1):

    return np.random.normal(0, stddev, size=(2*N))
```

[7]:
```
# test it
x = random_states(N, 5)
params = random_params_mf(N)
logpsi_mf(params, x)
```

[7]:
```
array([ 0.17967098,  0.20652151,  0.17270972,  0.08493403, -0.0767841 ])
```

[8]:
```
# check the shape
assert logpsi_mf(params, x).shape == (len(x),)
```

0.2.2 b) Gradient

[9]:
```
def grad_logpsi_mf(params, s):
    Ns, N = s.shape
    g = np.zeros((Ns,) + params.shape)
    g[:, :N] = (s == -1)
    g[:, N:] = (s == +1)
    return g
```

[10]:
```
grad_logpsi_mf(params, x)
```

[10]:
```
array([[1., 1., 0., 1., 0., 0., 1., 0., 1., 1., 1., 1., 1., 0., 0., 1.,
        0., 0., 1., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 1., 1., 0.],
       [1., 0., 1., 1., 1., 1., 0., 0., 1., 1., 0., 1., 1., 1., 1., 0.,
        0., 1., 0., 0., 0., 0., 1., 1., 0., 0., 1., 0., 0., 0., 0., 1.],
       [0., 1., 1., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 1., 0., 1.,
        1., 0., 0., 0., 1., 1., 0., 1., 1., 1., 1., 0., 1., 0., 1., 0.],
       [0., 0., 1., 0., 1., 1., 0., 1., 0., 0., 1., 1., 1., 1., 0., 0.,
        1., 1., 0., 1., 0., 0., 1., 0., 1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 1., 0., 0., 0.,
        0., 1., 1., 1., 1., 1., 0., 0., 0., 1., 1., 1., 0., 1., 1., 1.]])
```

[11]:
```
assert grad_logpsi_mf(params, x).shape == (len(x),)+params.shape
```

0.2.3 c) Direct sampling

[12]:
```
def sample_direct_mf(logpsi, params, N, N_samples):
    assert logpsi == logpsi_mf
    p_down = np.exp(2 * params[:N])
    p_up = np.exp(2 * params[N:])
    pdf = p_up / (p_up + p_down)
      = np.random.uniform(size=(N_samples, N))
    samples = (pdf> ) * (+1) + (1 - (pdf> )) * (-1)
    return samples
```

[13]:
```
samples_direct = sample_direct_mf(logpsi_mf, params, N, Ns)
samples_direct
```

[13]:
```
array([[ 1,  1, -1, …, -1, -1, -1],
       [-1,  1,  1, …, -1,  1, -1],
       [ 1,  1, -1, …, -1,  1,  1],
       …,
       [-1, -1, -1, …, -1, -1, -1],
       [ 1, -1,  1, …,  1, -1, -1],
       [-1,  1,  1, …, -1,  1,  1]])
```

[14]:
```
assert samples_direct.shape == (Ns, N)
```

0.2.4 d) Monte Carlo Sampling

[15]:
```
# we need a funtion to do the update move to obtain new configurations
# we will use this as propose_fn below

def single_spin_flip(x):
    assert x.ndim == 1
    N, = x.shape
```

```
    x = x.copy()
    i = np.random.randint(0, N)
    x[i] *= -1
    return x
```

[16]:
```
s = np.array([-1,1,1,-1])

print(s)
print(single_spin_flip(s))
```

```
[-1  1  1 -1]
[-1  1  1  1]
```

[17]:
```
def sample_step(logpsi, params, x, logpsi_x, propose_fn):
    """
    One sampling step of Monte Carlo Markov Chain.
    logpsi: function giving the log wave-funtion of a batch of sample given a␣
 ↪set of parametes
            (params,x) -> log( (x))
    params: parameters of the ansatz
    x: sample on which to do a step (shape (N,) )
    logpsi_x: log-value of the wave-function for x
    propose_fn: update move
    """

    if logpsi_x is None:
        # for the sampling we work with a single sample
        # but our ansatz only supports batches
        # so we add a dummy batch dimension here
        logpsi_x = logpsi(params, np.expand_dims(x, 0))[0]

    # propose a new state
    x_proposed = propose_fn(x) # this function samples from T(x -> x')
    logpsi_x_proposed = logpsi(params, np.expand_dims(x_proposed, 0))[0]

    R = np.exp(2 * (logpsi_x_proposed - logpsi_x))
    u = np.random.uniform()
    accept = R > u

    if accept:
        return x_proposed, logpsi_x_proposed
    else:
        return x, logpsi_x
```

[18]:
```
def sample_mcmc(logpsi, params, N, N_samples, N_discard, x0=None,␣
 ↪propose_fn=single_spin_flip):
    """
```

```
    Monte Carlo Markov Chains sampling. Given an ansatz, it samples randomly␣
↪N_samples.
    logpsi: function giving the log wave-funtion of a batch of sample given a␣
↪set of parametes
            (params,x) -> log( (x))
    params: parameters of the ansatz
    N: number of sites
    N_samples: number of samples to generate
    N_discard: number of initial samples to discard (thermalization)
    x0: initial configuration
        (if None, a random sample is drawn from the Hilbert space)
    propose_fn: update move
    """

    # Initialization
    if x0 is None:
        x0 = random_states(N, 1)[0]

    x = x0
    logpsi_x = None

    # Thermalization : we don't keep the samples
    for i in range(N_discard):
        x, logpsi_x = sample_step(logpsi, params, x, logpsi_x, propose_fn)

    # MCMC
    samples = []
    for i in range(N_samples):
        x, logpsi_x = sample_step(logpsi, params, x, logpsi_x, propose_fn)
        samples.append(x)

    return np.vstack(samples)
```

[19]:
```
samples_mcmc = sample_mcmc(logpsi_mf, params, N, Ns, N_discard)
samples_mcmc
```

[19]:
```
array([[-1, -1,  1, …, -1,  1, -1],
       [-1,  1,  1, …, -1,  1, -1],
       [-1,  1,  1, …, -1,  1, -1],
       …,
       [-1,  1, -1, …, -1,  1, -1],
       [-1,  1, -1, …, -1,  1, -1],
       [-1,  1, -1, …, -1,  1, -1]])
```

[20]:
```
assert samples_mcmc.shape == (Ns, N)
```

0.2.5 e) Connected Elements

[21]:
```
def ising_hamiltonian(x, Γ=1, J=1):
    """
    TFI hamiltonian in 1D with pbc. Given a configuration x compute the all␣
 ↪connected x' and matrix elements Hxx' s.t. Hxx' != 0
    """

    # x is again a batch of samples:
    n_samples, n_sites = x.shape

    # there are n_sites + 1 connected states
    n_conn = n_sites + 1

    # intitalize arrays
    x_prime = np.zeros((n_samples, n_conn, n_sites), dtype=x.dtype)
    mels = np.zeros((n_samples, n_conn))

    # states

    # diagonal: we take the first row to be the diagonal where x==x'
    # (this is arbitrary, we could have picked any other order)
    x_prime[:, 0] = x

    # off-diagonal:
    # the remaining rows are the off-diagonal terms
    # We have to flip one spin each, we do this in order
    flip_mask = np.eye(n_sites, dtype=int) # where to flip
    x_prime[:, 1:] = (1-flip_mask) * np.expand_dims(x, 1)  + flip_mask * (- np.
 ↪expand_dims(x, 1))

    # matrix elements

    # diagonal
    mels[:, 0] = J * (x * np.roll(x, 1, axis=-1)).sum(axis=-1)

    # off-diagonal
    mels[:, 1:] = -Γ

    return x_prime, mels
```

[22]:
```
xp, mels = ising_hamiltonian(x, 1, 1)
```

[23]:
```
xp
```

[23]:
```
array([[[-1, -1,  1, …,  1,  1, -1],
        [ 1, -1,  1, …,  1,  1, -1],
```

```
        [-1,  1,  1, …,  1,  1, -1],
        …,
        [-1, -1,  1, …, -1,  1, -1],
        [-1, -1,  1, …,  1, -1, -1],
        [-1, -1,  1, …,  1,  1,  1]],

       [[-1,  1, -1, …, -1, -1,  1],
        [ 1,  1, -1, …, -1, -1,  1],
        [-1, -1, -1, …, -1, -1,  1],
        …,
        [-1,  1, -1, …,  1, -1,  1],
        [-1,  1, -1, …, -1,  1,  1],
        [-1,  1, -1, …, -1, -1, -1]],

       [[ 1, -1, -1, …, -1,  1, -1],
        [-1, -1, -1, …, -1,  1, -1],
        [ 1,  1, -1, …, -1,  1, -1],
        …,
        [ 1, -1, -1, …,  1,  1, -1],
        [ 1, -1, -1, …, -1, -1, -1],
        [ 1, -1, -1, …, -1,  1,  1]],

       [[ 1,  1, -1, …, -1,  1,  1],
        [-1,  1, -1, …, -1,  1,  1],
        [ 1, -1, -1, …, -1,  1,  1],
        …,
        [ 1,  1, -1, …,  1,  1,  1],
        [ 1,  1, -1, …, -1, -1,  1],
        [ 1,  1, -1, …, -1,  1, -1]],

       [[-1,  1,  1, …,  1,  1,  1],
        [ 1,  1,  1, …,  1,  1,  1],
        [-1, -1,  1, …,  1,  1,  1],
        …,
        [-1,  1,  1, …, -1,  1,  1],
        [-1,  1,  1, …,  1, -1,  1],
        [-1,  1,  1, …,  1,  1, -1]]])
```

[24]:
```
mels
```

[24]:
```
array([[ 0., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
        -1., -1., -1., -1.],
       [ 0., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
        -1., -1., -1., -1.],
       [-4., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
        -1., -1., -1., -1.],
       [ 0., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
```

```
        -1., -1., -1., -1.],
       [ 4., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
        -1., -1., -1., -1.]])
```

[25]:
```
assert xp.shape == (len(x), N+1, N)
assert mels.shape == (len(x), N+1)
```

0.2.6 f) Local operator

[26]:
```
def compute_eloc(operator, logpsi, params, x):
    Ns, N = x.shape
    xp, mels = operator(x)
    logpsi_x = logpsi(params, x)
    # logpsi can only take batches of samples (with one single batch dimension)
    # so we have to flatten the input and unflatten the output
    logpsi_xp = logpsi(params, xp.reshape(-1, N)).reshape(xp.shape[:-1])
    eloc = (mels * np.exp(logpsi_xp-np.expand_dims(logpsi_x, -1))).sum(axis=-1)
    return eloc
```

[27]:
```
eloc = compute_eloc(ising_hamiltonian, logpsi_mf, params, x)
```

[28]:
```
assert eloc.shape == (len(x),)
```

0.2.7 g+h) Stochastic estimates

[29]:
```
def expect(operator, logpsi, params, x):
    "compute the expectation value of an operator given a batch of samples"
    eloc = compute_eloc(operator, logpsi, params, x)
    E = eloc.mean()
    return E
```

[30]:
```
expect(ising_hamiltonian, logpsi_mf, params, samples_direct)
```

[30]:
```
-15.965010091574
```

[31]:
```
expect(ising_hamiltonian, logpsi_mf, params, samples_mcmc)
```

[31]:
```
-15.920906915118627
```

0.2.8 i) Gradient descent

[32]:
```
def expect_and_grad(operator, logpsi, grad_logpsi, params, x):

    eloc = compute_eloc(operator, logpsi, params, x)
    E = eloc.mean()
    Dk = grad_logpsi(params, x)
    delta_eloc = eloc - eloc.mean(axis=0, keepdims=True)
```

```
    Gk = 2 * np.expand_dims(delta_eloc, 1) * Dk
    grad = Gk.mean(axis=0)
    return E, grad
```

[33]:
```
_, g = expect_and_grad(ising_hamiltonian, logpsi_mf, grad_logpsi_mf, params, x)
assert g.shape == params.shape
```

[34]:
```
# we will pass this as opt_fn to the vmc function below
def sgd(params, grad,  ):
    return params -   * grad
```

[35]:
```
def vmc(operator, sample_fn, opt_fn, logpsi, grad_logpsi, params, N, nsteps):

    energies = []

    for i in tqdm(range(nsteps)):

        # sample
        x = sample_fn(logpsi, params, N)
        # estimate energy and gradients
        E, grad = expect_and_grad(operator, logpsi, grad_logpsi, params, x)
        energies.append(E)
        # compute updated parameters
        params = opt_fn(params, grad)

    E = expect(operator, logpsi, params, x)
    energies.append(E)

    return params, energies
```

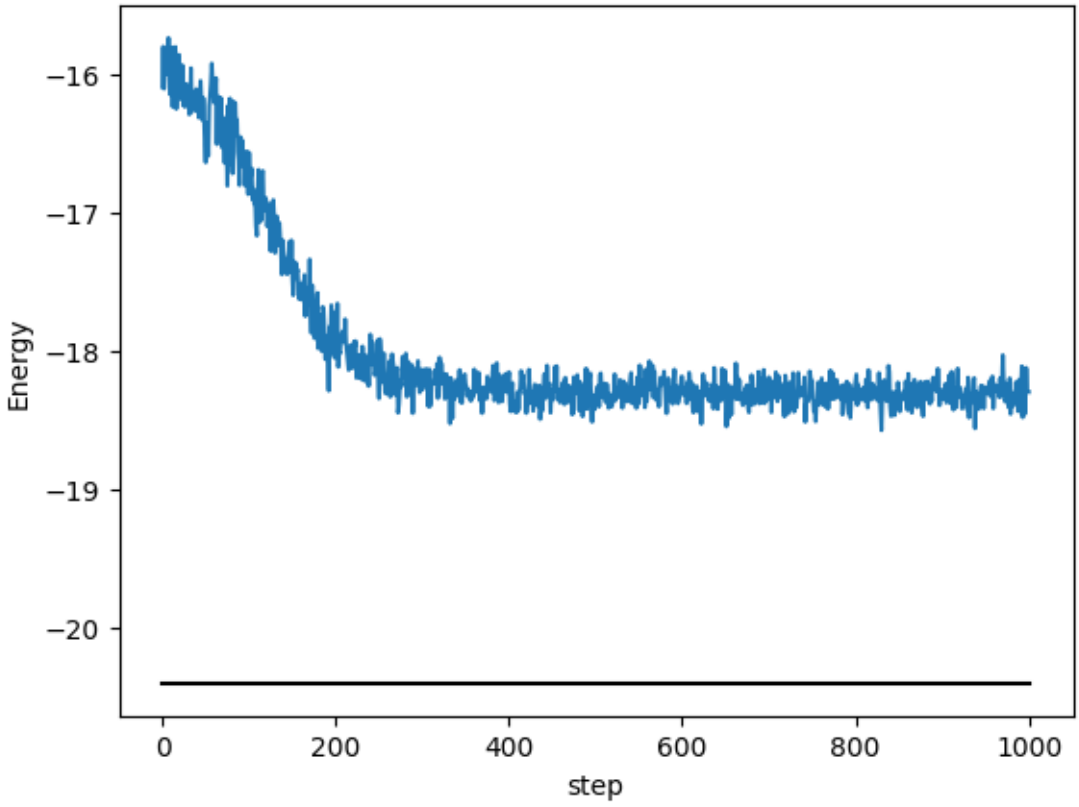
Mean-field Ansatz with direct sampling

[36]:
```
# setup the direct sampler fixing the number of samples
sa = partial(sample_direct_mf, N_samples=Ns)

# set up the optimizer by fixing the learning rate
opt = partial(sgd,  =0.01)

#set up the operator
ha = partial(ising_hamiltonian, Γ=1, J=1)

lpsi = logpsi_mf
glpsi = grad_logpsi_mf
# set up the initial guess for the parameters
par = random_params_mf(N, 0.1)

_, energies = vmc(ha, sa, opt, lpsi, glpsi, par, N, 1000)
plt.figure()
```

```
plt.plot(energies)
E0_analytical = ising1d_energy(N, 1)
plt.plot([0, len(energies)], [E0_analytical,]*2, color='k')
plt.xlabel('step')
plt.ylabel(('Energy'))
print('relative error: {:0.2e}'.format(err_rel(E0_analytical, energies[-1])))
plt.show()
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]
```

```
relative error: 1.15e-01
```

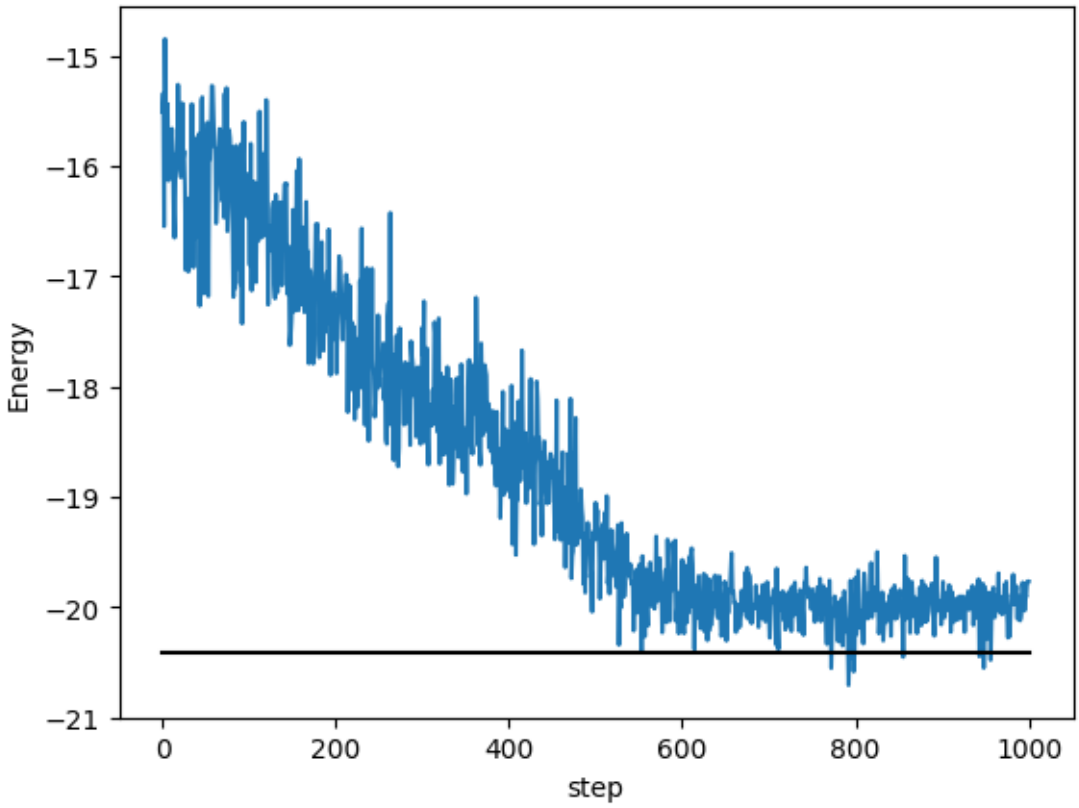


Mean-field with MCMC sampling

[37]:
```
sa = partial(sample_mcmc, N_samples=Ns, N_discard=N_discard)
opt = partial(sgd,  =0.01)
ha = partial(ising_hamiltonian, Γ=1, J=1)

lpsi = logpsi_mf
glpsi = grad_logpsi_mf
par = random_params_mf(N, 0.1)
```

```
_, energies = vmc(ha, sa, opt, lpsi, glpsi, par, N, 1000)

plt.figure()
plt.plot(energies)
E0_analytical = ising1d_energy(N, 1)
plt.plot([0,  len(energies)], [E0_analytical,]*2, color='k')
plt.xlabel('step')
plt.ylabel(('Energy'))
print('relative error: {:0.2e}'.format(err_rel(E0_analytical, energies[-1])))
plt.show()
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]
relative error: 3.22e-02
```

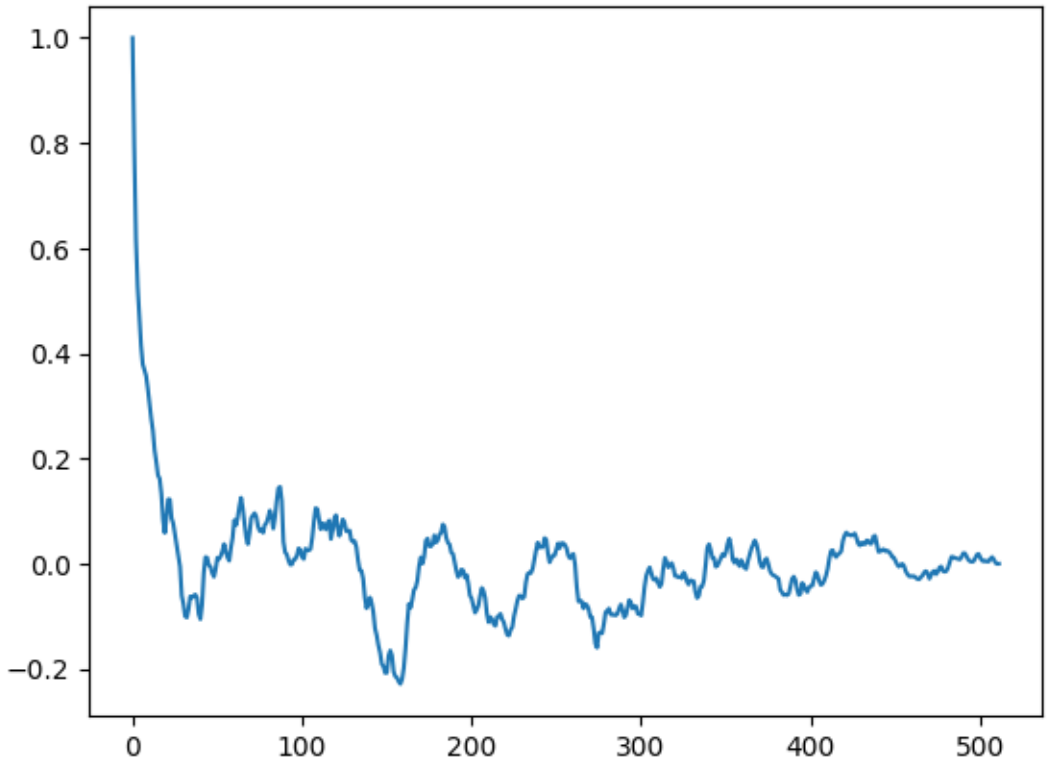


0.2.9 j) Auto-correlation time (bonus)

[38]:
```
def corr_fn_fft(g):
    n = len(g)
    A = np.fft.rfft(g - g.mean(axis=0), n=2*n)
    B = A * A.conj()  / len(g)
    return np.fft.irfft(B)[:n] / ((g**2).mean(axis=0) - g.mean(axis=0)**2)
```

[39]:
```
par_mf = random_params_mf(N, 0.1)
x = sample_mcmc(logpsi_mf, par_mf, N_samples=Ns, N=N, N_discard=N_discard)
eloc = compute_eloc(ising_hamiltonian, logpsi_mf, par_mf, x)
```

[40]:
```
  = corr_fn_fft(eloc)
plt.figure()
plt.plot( )
plt.show()
```



[41]:
```
jcut = np.argmin(  > 0)
  = 0.5 + ( [:jcut]).sum()
```

[41]:
```
8.19596157065558
```

0.3 Exercise 7.2

0.3.1 a) Nearest-neighbour Jastrow

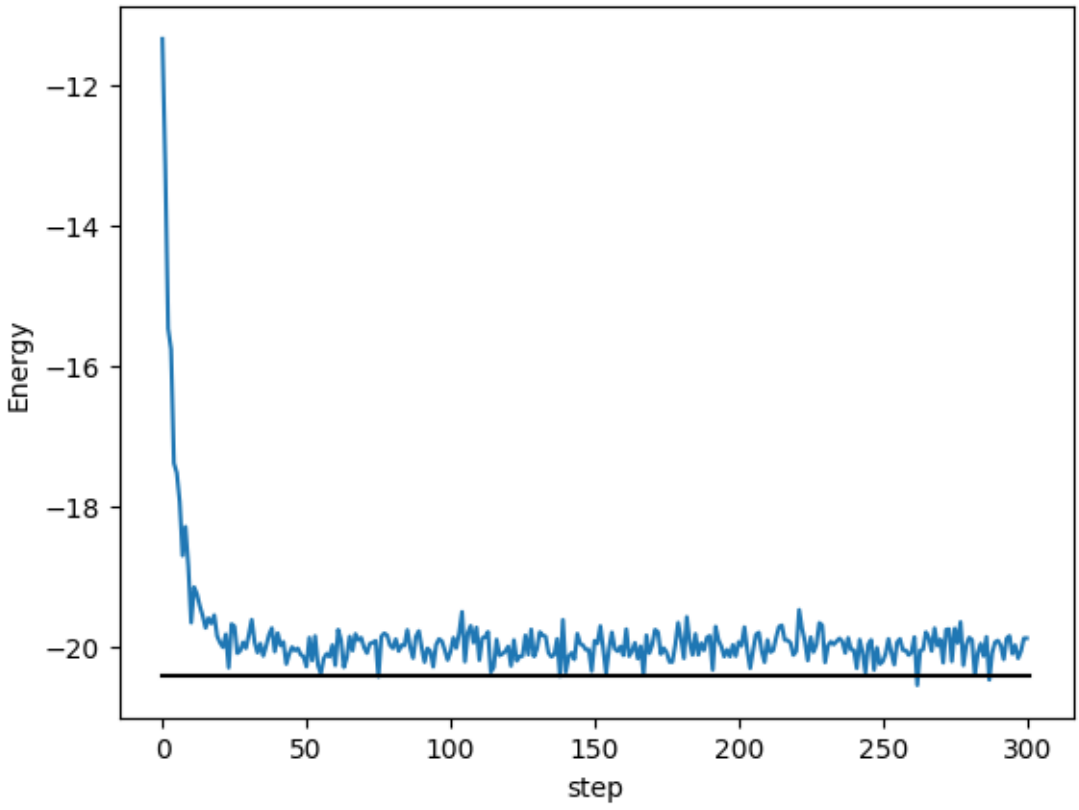
[42]:
```
def logpsi_jastrow_nearest(params, s):
    Ns, N = s.shape
    J1 = params
    return J1 * (s * np.roll(s, -1, axis=-1)).sum(axis=-1) # nearest neighbors␣
 ↪(pbc)

def grad_logpsi_jastrow_nearest(params, s):
    Ns, N = s.shape
    return (s * np.roll(s, -1, axis=-1)).sum(axis=-1, keepdims=True) # nearest␣
 ↪neighbors (pbc)

def random_params_jastrow_nearest(N, stddev=0.1):
    return np.random.normal(0, stddev, size=1)
```

[43]:
```
# check
params_jastrow = random_params_jastrow_nearest(N, 0.1)
assert logpsi_jastrow_nearest(params_jastrow, x).shape == (len(x),)
assert grad_logpsi_jastrow_nearest(params_jastrow, x).shape ==␣
 ↪(len(x),len(params_jastrow))
```

[44]:
```
sa = partial(sample_mcmc, N_samples=Ns, N_discard=N_discard)
opt = partial(sgd,  =0.001) # needs a fairly small learning rate
ha = partial(ising_hamiltonian, Γ=1, J=1)

lpsi = logpsi_jastrow_nearest
glpsi = grad_logpsi_jastrow_nearest
par = random_params_jastrow_nearest(N, 0.1)

_, energies = vmc(ha, sa, opt, lpsi, glpsi, par, N, 300)
plt.figure()
plt.plot(energies)
E0_analytical = ising1d_energy(N, 1)
plt.plot([0,  len(energies)], [E0_analytical,]*2, color='k')
plt.xlabel('step')
plt.ylabel(('Energy'))
print('relative error: {:0.2e}'.format(err_rel(E0_analytical, energies[-1])))
plt.show()
```

```
  0%|          | 0/300 [00:00<?, ?it/s]
```

```
relative error: 2.63e-02
```



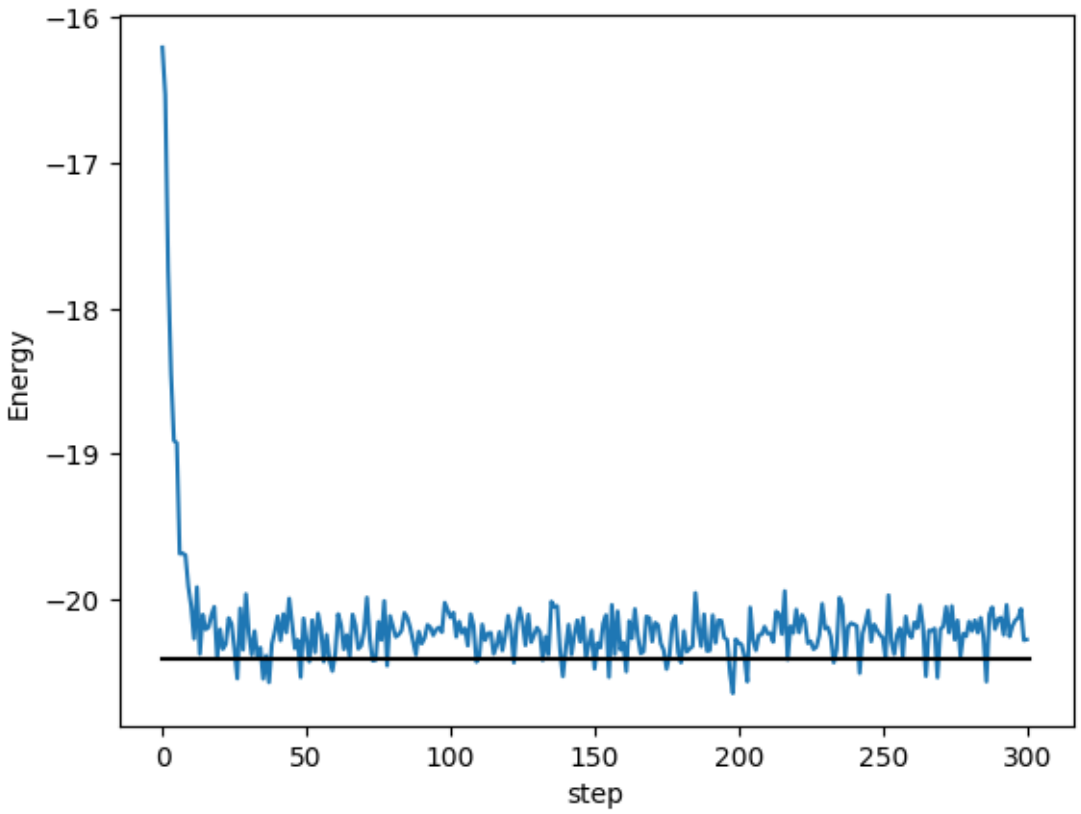
0.3.2 b) Nearest+next-nearest-neighbour Jastrow

[45]:
```
def logpsi_jastrow_next_nearest(params, s):
    Ns, N = s.shape
    J1 = params[0]
    J2 = params[1]
    res1 = logpsi_jastrow_nearest(J1, s)
    res2 = J2 * (s * np.roll(s, -2, axis=-1)).sum(axis=-1) # next-nearest␣
 ↪neighbors (pbc)
    return res1+res2

def grad_logpsi_jastrow_next_nearest(params, s):
    Ns, N = s.shape
    J1 = params[0]
    # J2 = params[1]
    g1 = grad_logpsi_jastrow_nearest(J1, s)
    g2 = (s * np.roll(s, -2, axis=-1)).sum(axis=-1, keepdims=True) #␣
 ↪next-nearest neighbors (pbc)
    return np.concatenate([g1,g2], axis=1)

def random_params_jastrow_next_nearest(N, stddev=0.1):
```

```
    return np.random.normal(0, stddev, size=2)
```

[46]:
```
sa = partial(sample_mcmc, N_samples=Ns, N_discard=N_discard)
opt = partial(sgd,  =0.001)
ha = partial(ising_hamiltonian, Γ=1, J=1)

lpsi = logpsi_jastrow_next_nearest
glpsi = grad_logpsi_jastrow_next_nearest
par = random_params_jastrow_next_nearest(N, 0.1)

_, energies = vmc(ha, sa, opt, lpsi, glpsi, par, N, 300)

plt.figure()
plt.plot(energies)
E0_analytical = ising1d_energy(N, 1)
plt.plot([0,  len(energies)], [E0_analytical,]*2, color='k')
plt.xlabel('step')
plt.ylabel(('Energy'))
print('relative error: {:0.2e}'.format(err_rel(E0_analytical, energies[-1])))
plt.show()
```

```
  0%|          | 0/300 [00:00<?, ?it/s]
relative error: 6.54e-03
```



0.3.3 c) Arbitrary n-neighbours Jastrow

[47]:
```
def logpsi_jastrow_n_neighbours(params, s):
    Ns, N = s.shape
    n = params.shape[0]

    res = 0
    for i in range(n):
        res += params[i]*(s * np.roll(s, -(i+1), axis=-1)).sum(axis=-1)

    return res

def grad_logpsi_jastrow_n_neighbours(params, s):
    Ns, N = s.shape
    n = params.shape[0]

    g = []
    for i in range(n):
        g.append( (s * np.roll(s, -(i+1), axis=-1)).sum(axis=-1, keepdims=True)␣
 ↪)

    return np.concatenate(g, axis=1)

def random_params_jastrow_n_neighbours(N, n=1, stddev=0.1):
    assert n>=1 and n<N
    return np.random.normal(0, stddev, size=n)
```

[48]:
```
sa = partial(sample_mcmc, N_samples=Ns, N_discard=N_discard)
opt = partial(sgd,  =0.001)
ha = partial(ising_hamiltonian, Γ=1, J=1)

E0_analytical = ising1d_energy(N, 1)

E_converged = []
delta_E = []
for n in range(1,N//2+1): #<- we only go up to N//2 since we have pbcs
    lpsi = logpsi_jastrow_n_neighbours
    glpsi = grad_logpsi_jastrow_n_neighbours
    par = random_params_jastrow_n_neighbours(N, n, 0.1)

    _, energies = vmc(ha, sa, opt, lpsi, glpsi, par, N, 300)

    E_converged.append( energies[-1] )
    delta_E.append( err_rel(E0_analytical, energies[-1]) )
```

```
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
0%|          | 0/300 [00:00<?, ?it/s]
```

[49]:
```
plt.figure()
plt.plot(np.arange(1,N//2+1), E_converged)
plt.plot([0,N//2+1], [E0_analytical,E0_analytical])
plt.xlabel('Number of neighbours')
plt.ylabel('Final Energy')
plt.show()
```