ex09_solutions

April 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 Solutions 09 Time-dependent variational principle

The solution of the exercises are self contained and can be run independentely from each others.

0.1.1 Exercise 9.1 - The Time Dependent Variational Principle (TDVP)

In this exercise we will use the Time Dependent Variational Principle to variationally approximate the dynamics of a Transverse Field Ising chain. We will compare the results obtained with those of Lesson 3.

a) Let's start with the simple 2 qubit case

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy.linalg import solve
from scipy.sparse import csr_array, identity, kron
from scipy.sparse.linalg import eigsh
```

[2]:
```
# Pauli matrices
sx = csr_array([[0, 1], [1, 0]])
sz = csr_array([[1, 0], [0, -1]])

spin_up = np.array([1, 0])
spin_down = np.array([0, 1])
```

[3]:
```
def pauli_op(pauli, i, N):
    left = identity(2**i)
    right = identity(2 ** (N - i - 1))
    mat = kron(kron(left, pauli), right)

    # Explicitly convert to CSR since kron likes to return COO
    mat = csr_array(mat)
    return mat
```

```
# Function to generate a list of Pauli operators
def list_of_op(pauli, n_spins):
    return [pauli_op(pauli, i, n_spins) for i in range(n_spins)]
```

[4]:
```
# Generate the Ising hamiltonian
def ising_hamiltonian(sx_list, sz_list, J, gamma):
    n_spins = len(sx_list)
    ham = 0
    for i in range(n_spins):
        ham -= gamma * sx_list[i]
    for i in range(n_spins - 1):
        ham += J * sz_list[i] @ sz_list[i + 1]
    return ham
```

[5]:
```
# Observable to measure during the time evolution


def Hz(n_spins, site):
    # Measure sigma_z on spin site
    sz = csr_array([[1, 0], [0, -1]])
    left = identity(2**site)
    right = identity(2 ** (n_spins - site - 1))
    mat = kron(kron(left, sz), right)

    mat = csr_array(mat)
    return mat


def Mz(n_spins):
    # Measure the mean magnetization along the z-axis
    mz = 0
    for i in range(n_spins):
        mz += Hz(n_spins, i)

    return (1 / n_spins) * mz
```

[6]:
```
# Function to measure expectation values
def measure(psi, operator):
    # First do M @ v, which is faster than v @ M for CSR matrix, then do v @ v
    return (psi.conj().T @ (operator @ psi)).real
```

For the simple variational form

$$|\psi(\theta(t))\rangle = e^{-i\theta(t)(\sigma_1^x+\sigma_2^x)}|\phi(0)\rangle \text{ with } |\phi(0)\rangle = |\downarrow\rangle \otimes |\downarrow\rangle$$

the S matrix is a scalar and reads

$$S = 2 + 2\langle\psi|X_1X_2|\psi\rangle - |\langle\psi|X_1 + X_2|\psi\rangle|^2$$

while

$$C = i\langle\psi|(X_1 + X_2)H|\psi\rangle - i\langle\psi|(X_1 + X_2)|\psi\rangle\langle\psi|H|\psi\rangle$$

So we need to measure those values and solve the equation of motion

[7]:
```
def psi_t( ):
    x1 = kron(sx, identity(2))
    x2 = kron(identity(2), sx)
    exp_op = (np.cos( ) * identity(4) - 1j * np.sin( ) * x1) @ (
        np.cos( ) * identity(4) - 1j * np.sin( ) * x2
    )

    psi0 = np.kron(spin_down, spin_down)

    return exp_op @ psi0
```

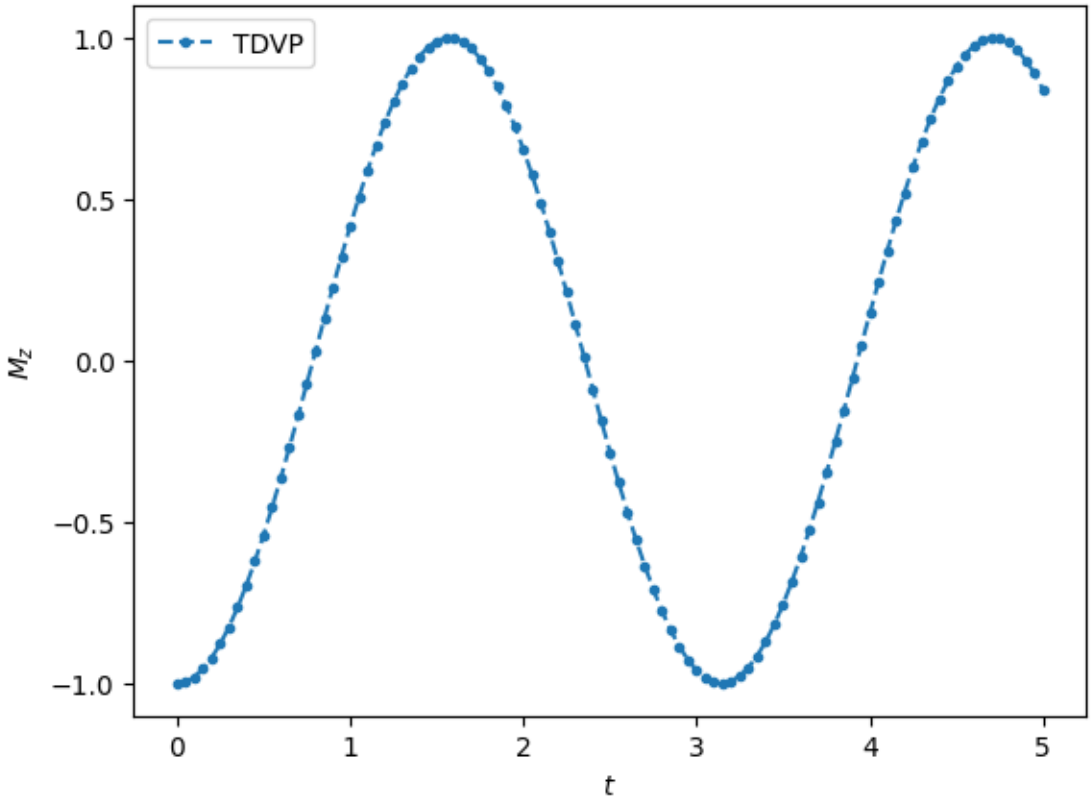
[8]:
```
# Create operators and Hamiltonian to measure
sx_list = list_of_op(sx, 2)
sz_list = list_of_op(sz, 2)

sum_sx = sx_list[0] + sx_list[1]
prod_x1x2 = sx_list[0] @ sx_list[1]
hami = ising_hamiltonian(sx_list, sz_list, J=0, gamma=1)
```

[9]:
```
# For every time step measure S and C and find \dot{\theta}
T = 5
nt = 100
dt = T / nt
ts = np.linspace(0, T, nt + 1)  # np.linspace is defined for [start, end]
```

[10]:
```
theta = 0
psit = psi_t(theta)

mz2 = Mz(2)
mz_2spin = [measure(psit, mz2)]

for step in range(nt):
    # 2) Create S and C
    S_mat = 2 + measure(psit, prod_x1x2) - measure(psit, sum_sx) ** 2
    C_vec = 1j * measure(psit, sum_sx @ hami) - 1j * measure(psit, sum_sx) *␣
 ↪measure(
        psit, hami
    )
```

```
    # 3) Solve the linear system
    _dot = np.imag(C_vec) / np.real(S_mat)

    # 4) Update the parameters
    theta = theta +  _dot * dt

    # 5) Create the new variational state and measure observables
    psit = psi_t(theta)

    mz_2spin.append(measure(psit, mz2))
```

[11]:
```
# Plot the results
plt.plot(ts, mz_2spin, linestyle="dashed", color="C0", marker=".", label="TDVP")
plt.ylabel(r"$M_z$")
plt.xlabel(r"$t$")
plt.yticks([-1, -0.5, 0, 0.5, 1])
plt.legend()
plt.show()
```



b) Now we need to create the mean-field ansatz

$$|\psi\left(\theta_1(t), \ldots, \theta_N(t)\right)\rangle = |\psi(t)\rangle = \prod_{i=1}^{N} e^{i\theta_i(t)X_i}|\psi(0)\rangle$$

given the initial state

$$|\psi(0)\rangle = \bigotimes_{i=1}^{N} |\downarrow\rangle\,,$$

where we indicated $X_i = \underbrace{\hat{I}\otimes\hat{I}\otimes\ldots\hat{I}}_{i-1 \text{ times}}\otimes\hat{\sigma}^x\otimes\underbrace{\hat{I}\otimes\cdots\otimes\hat{I}}_{N-i \text{ times}}$.

We will define it as a list of operators acting on the state $|\psi(0)\rangle$.

[12]:
```
def mf_ansatz_op(n_spins, _vec):
    # Generate the operators associated to the mean-field ansatz
    # given a vector of parameters _vec
    op_list = []

    for i in range(len(_vec)):
        left = identity(2**i)
        right = identity(2 ** (N - i - 1))
        mat = kron(kron(left, sx), right)

        I = identity(2**N)
        mat = np.cos(_vec[i]) * I - 1j * np.sin(_vec[i]) * mat
        mat = csr_array(mat)
        op_list.append(mat)
    return op_list


def mf_ansatz_psi(ansatz_op, psi):
    for mat in ansatz_op:
        psi = mat @ psi

    return psi


def squared_norm(psi):
    return (psi.conj().T @ psi).real
```

Now we have to define the method to compute the S matrix and the C vector.

For the real time evolution we will need the real part of S and the imaginary part of C.

For this simple ansatz we can explicitly calculate the derivative with respect to θ_i and therefore give the expression of S and C as expectation values calculated over the ansatz state.

Indeed, given:

$$\frac{\partial}{\partial\theta_i}|\psi(t)\rangle = |\partial_i\psi(t)\rangle = i\sigma_x^i|\psi(t)\rangle$$

we have

$$S_{ij} = \langle\partial_i\psi|\partial_j\psi\rangle/\langle\psi|\psi\rangle - \langle\partial_i\psi|\psi\rangle\langle\psi|\partial_j\psi\rangle/\langle\psi|\psi\rangle^2 = \langle\psi|X_iX_j|\psi\rangle/\langle\psi|\psi\rangle - \langle\psi|X_i|\psi\rangle\langle\psi|X_j|\psi\rangle/\langle\psi|\psi\rangle^2$$

and

$$C_k = -i\langle\psi|X_kH|\psi\rangle/\langle\psi|\psi\rangle + i\langle\psi|X_k|\psi\rangle\langle\psi|H|\psi\rangle/\langle\psi|\psi\rangle^2$$

Therefore, for each time step it suffices to measure a series of expectation values of X_j and H and use those values to evaluate the derivatives of the parameters using

$$S^R\dot{\theta} = C^I$$

[13]:
```
# Define the simulation time
T = 5
nt = 100
dt = T / nt
ts = np.linspace(0, T, nt + 1)  # np.linspace is defined for [start, end]
```

- Case $J = 0$

[14]:
```
# Define the system
N = 10
J = 0.0
Gamma = 1.0
```

[15]:
```
# Create the list of operators to be measured
sx_list = list_of_op(sx, N)
sz_list = list_of_op(sz, N)

sxsx_list = []
for i in range(N):
    sxsx_row = []
    for j in range(N):
        sxsx_row.append(sx_list[i] @ sx_list[j])
    sxsx_list.append(sxsx_row)

# Hamiltonian
hami = ising_hamiltonian(sx_list, sz_list, J, Gamma)
sx_hami_list = []
for sx_i in sx_list:
    sx_hami_list.append(sx_i @ hami)
```

[16]:
```
# Function to measure expectation values
def measure(psi, operator):
    return (psi.conj().T @ (operator @ psi)).real
```

Now define the routine for the TDVP

[17]:
```
def S( _vec, sx_meas, sxsx_meas, psi_norm):
    nparams = len( _vec)
    s_mat = np.zeros((nparams, nparams), dtype=complex)

    # Create the S matrix
    for i in range(nparams):
        for j in range(nparams):
            s_mat[i, j] = sxsx_meas[i, j] / psi_norm - sx_meas[i] * sx_meas[j] /
 ↪ psi_norm**2

    return s_mat


def C( _vec, hami_meas, sx_meas, sx_hami_meas, psi_norm):
    nparams = len( _vec)
    c_vec = np.zeros(nparams, dtype=complex)

    for i in range(nparams):
        c_vec[i] = -1j * sx_hami_meas[i] / psi_norm + 1j * sx_meas[i] *␣
 ↪hami_meas  / psi_norm**2

    return c_vec
```

Now define all the quantities useful for the TDVP loop

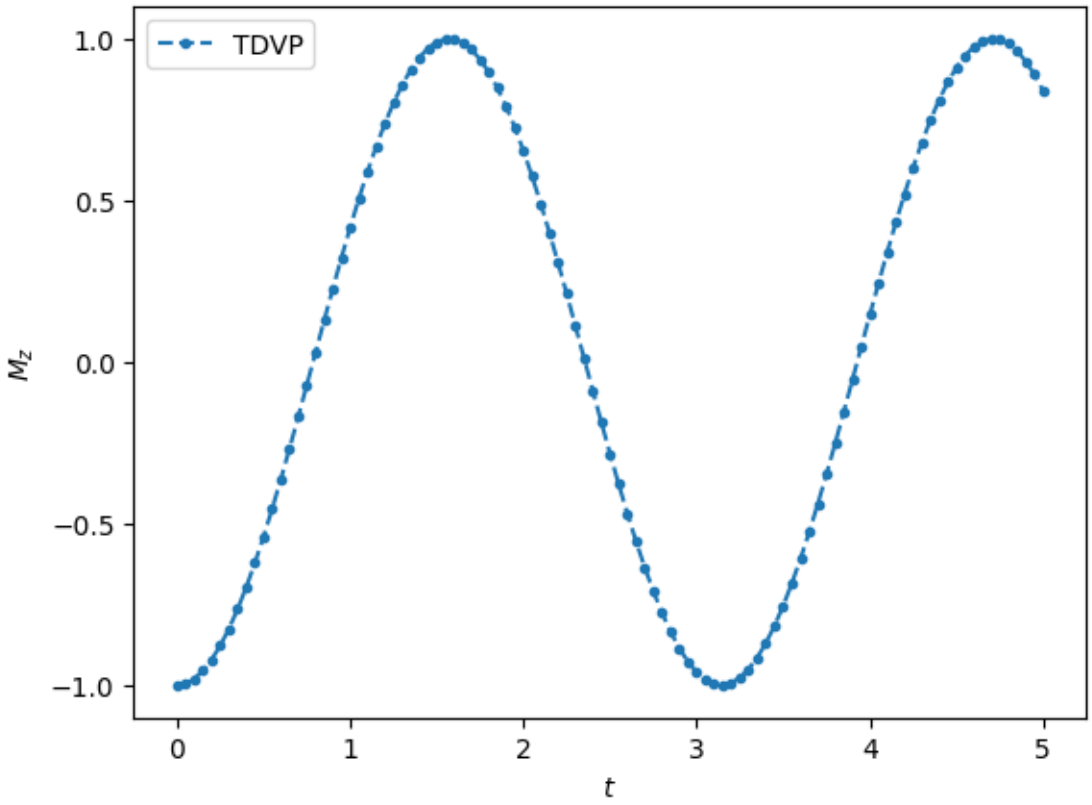
[18]:
```
# Initialise the TDVP loop

 _params = np.zeros(N)

# Initial state
psi0 = 1
for i in range(N):
    psi0 = np.kron(psi0, spin_down)

# And initial variational ansatz
psi_op = mf_ansatz_op(N,  _params)
psi_var = mf_ansatz_psi(psi_op, psi0)
psi_norm = squared_norm(psi_var)

# Observables to save
# Magnetization
mz_op = Mz(N)
```

```
mz_tdvp = []
mz_tdvp.append(measure(psi0, mz_op))

# Energy
e_tdvp = []
e_tdvp.append(measure(psi0, hami))
```

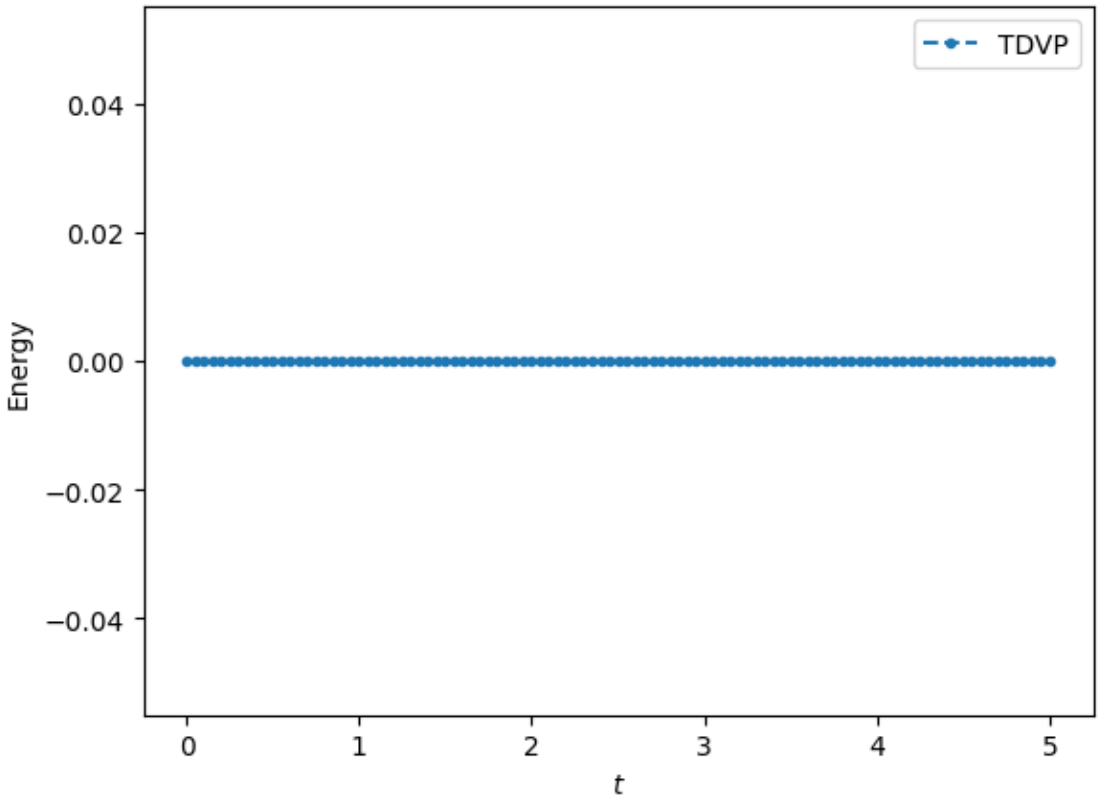
TDVP loop

[19]:
```
for step in range(nt):
    # 1) Measure all the operators to evaluate S and C
    sx_meas = np.asarray([measure(psi_var, x) for x in sx_list])
    sxsx_meas = np.asarray(
        [[measure(psi_var, x) for x in sxsx_row] for sxsx_row in sxsx_list]
    )
    energy = measure(psi_var, hami)
    sx_hami_meas = np.asarray([measure(psi_var, x) for x in sx_hami_list])

    # 2) Create S and C
    S_mat = S( _params, sx_meas, sxsx_meas, psi_norm)
    C_vec = C( _params, energy, sx_meas, sx_hami_meas, psi_norm)

    # 3) Solve the linear system
     _dot = solve(np.real(S_mat), np.imag(C_vec))

    # 4) Update the parameters
     _params =  _params +  _dot * dt

    # 5) Create the new variational state and measure observables
    psi_op = mf_ansatz_op(N,  _params)
    psi_var = mf_ansatz_psi(psi_op, psi0)
    psi_norm = squared_norm(psi_var)

    mz_tdvp.append(measure(psi_var, mz_op))
    e_tdvp.append(measure(psi_var, hami))
```

[20]:
```
# Plot the results
plt.plot(ts, mz_tdvp, linestyle="dashed", color="C0", marker=".", label="TDVP")
plt.ylabel(r"$M_z$")
plt.xlabel(r"$t$")
plt.yticks([-1, -0.5, 0, 0.5, 1])
plt.legend()
plt.show()
```



[21]:
```
# Check also the energy --- it has to remain constant during time evolution
plt.plot(ts, e_tdvp, linestyle="dashed", color="C0", marker=".", label="TDVP")
plt.ylabel(r"Energy")
plt.xlabel(r"$t$")
plt.legend()
plt.show()
```



Now we recall the time evolution algorithm from exercise session 3, to compare the results

[22]:
```
def exp_Hx(N, i, delta):
    sx = csr_array([[0, 1], [1, 0]])
    left = identity(2**i)
    right = identity(2 ** (N - i - 1))
    mat = kron(kron(left, sx), right)

    I = identity(2**N)
    mat = np.cos(delta / 2) * I - 1j * np.sin(delta / 2) * mat
    mat = csr_array(mat)
    return mat


def exp_Hzz(N, i, delta):
    sz = csr_array([[1, 0], [0, -1]])

    left = identity(2**i)
    right = identity(2 ** (N - i - 2))
    mat = kron(kron(left, kron(sz, sz)), right)
```

```
    I = identity(2**N)
    mat = np.cos(delta / 2) * I - 1j * np.sin(delta / 2) * mat
    mat = csr_array(mat)
    return mat


# Generate all time evolution operators
def gen_exp_hams(N, J, Gamma, dt):
    exp_Hxs = [exp_Hx(N, i, -Gamma * dt) for i in range(N)]
    exp_Hzzs = [exp_Hzz(N, i, J * dt) for i in range(N - 1)]
    exp_Hzz_halfs = [exp_Hzz(N, i, J / 2 * dt) for i in range(N - 1)]
    return exp_Hxs, exp_Hzzs, exp_Hzz_halfs
```

[23]:
```
def time_evol(exp_Hxs, exp_Hzzs, exp_Hzz_halfs, psi, t, dt):
    n_steps = round(t / dt)
    for mat in exp_Hzz_halfs:
        psi = mat @ psi
    for mat in exp_Hxs:
        psi = mat @ psi
    for step in range(n_steps):
        for mat in exp_Hzzs:
            psi = mat @ psi
        for mat in exp_Hxs:
            psi = mat @ psi
    for mat in exp_Hzz_halfs:
        psi = mat @ psi
    return psi
```

- Case $J = 0$

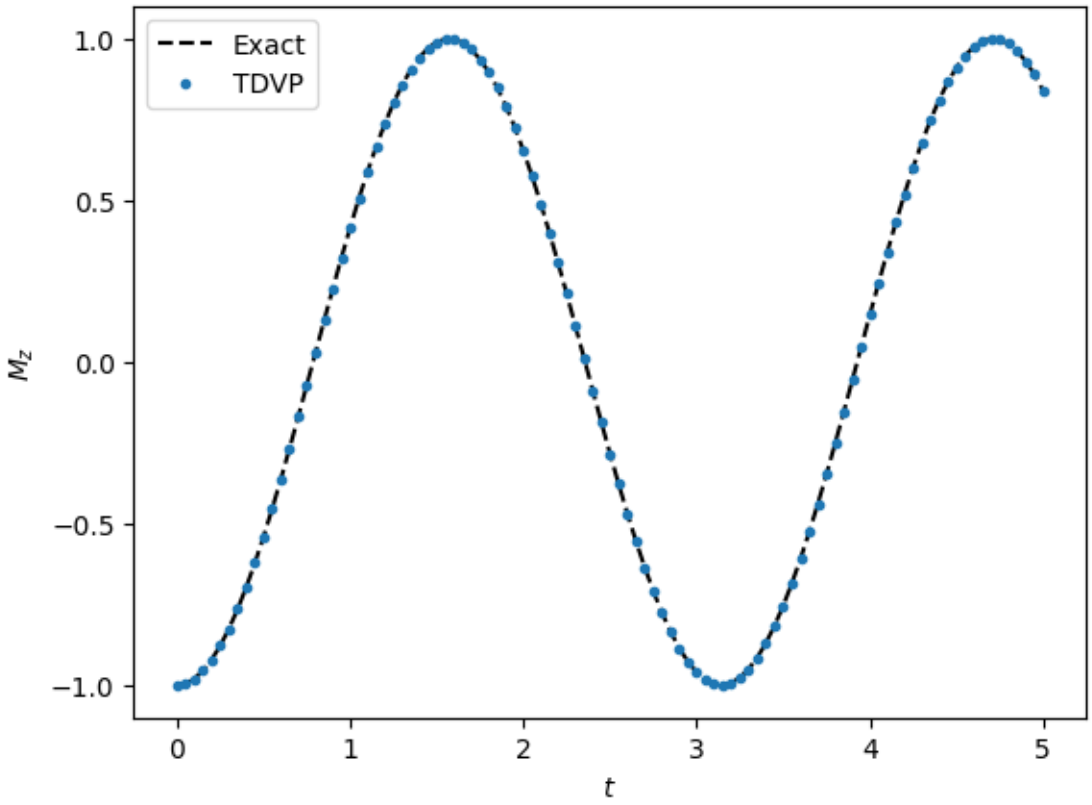
[24]:
```
# Hamiltonian parameters
N = 10
J = 0.0
Gamma = 1.0
```

[25]:
```
# Initial state
psi_ex = 1
for i in range(N):
    psi_ex = np.kron(psi_ex, spin_down)

# Magnetization operator
mz_op = Mz(N)
```

[26]:
```
# Evolve and measure the magnetization

exp_Hxs, exp_Hzzs, exp_Hzz_halfs = gen_exp_hams(N, J=J, Gamma=Gamma, dt=dt)
mz_t = np.empty(nt + 1)
```

```
mz_t[0] = measure(psi_ex, mz_op)

for i in range(1, nt + 1):
    # Every call evolves by a time-step dt
    psi_ex = time_evol(exp_Hxs, exp_Hzzs, exp_Hzz_halfs, psi_ex, dt, dt)
    mz_t[i] = measure(psi_ex, mz_op)
```

[27]:
```
# Plot the magnetization as a function of time
plt.plot(ts, mz_t, linestyle="dashed", color="black", label="Exact")
plt.plot(ts, mz_tdvp, linestyle="", color="C0", marker=".", label="TDVP")
plt.ylabel(r"$M_z$")
plt.xlabel(r"$t$")
plt.yticks([-1, -0.5, 0, 0.5, 1])
plt.legend()
plt.show()
```



- Case $J = 0.25$

[28]:
```
# Hamiltonian parameters
J = 0.25
Gamma = 1.0
```

```
hami = ising_hamiltonian(sx_list, sz_list, J, Gamma)
```

[29]:
```
# Exact

# Initial state
psi_ex = 1
for i in range(N):
    psi_ex = np.kron(psi_ex, spin_down)

# Magnetization operator
mz_op = Mz(N)

exp_Hxs, exp_Hzzs, exp_Hzz_halfs = gen_exp_hams(N, J=J, Gamma=Gamma, dt=dt)
mz_t = np.empty(nt + 1)

mz_t[0] = measure(psi_ex, mz_op)
e_t = [measure(psi_ex, hami)]

for i in range(1, nt + 1):
    # Every call evolves by a time-step dt
    psi_ex = time_evol(exp_Hxs, exp_Hzzs, exp_Hzz_halfs, psi_ex, dt, dt)
    mz_t[i] = measure(psi_ex, mz_op)
    e_t.append(measure(psi_ex, hami))
```

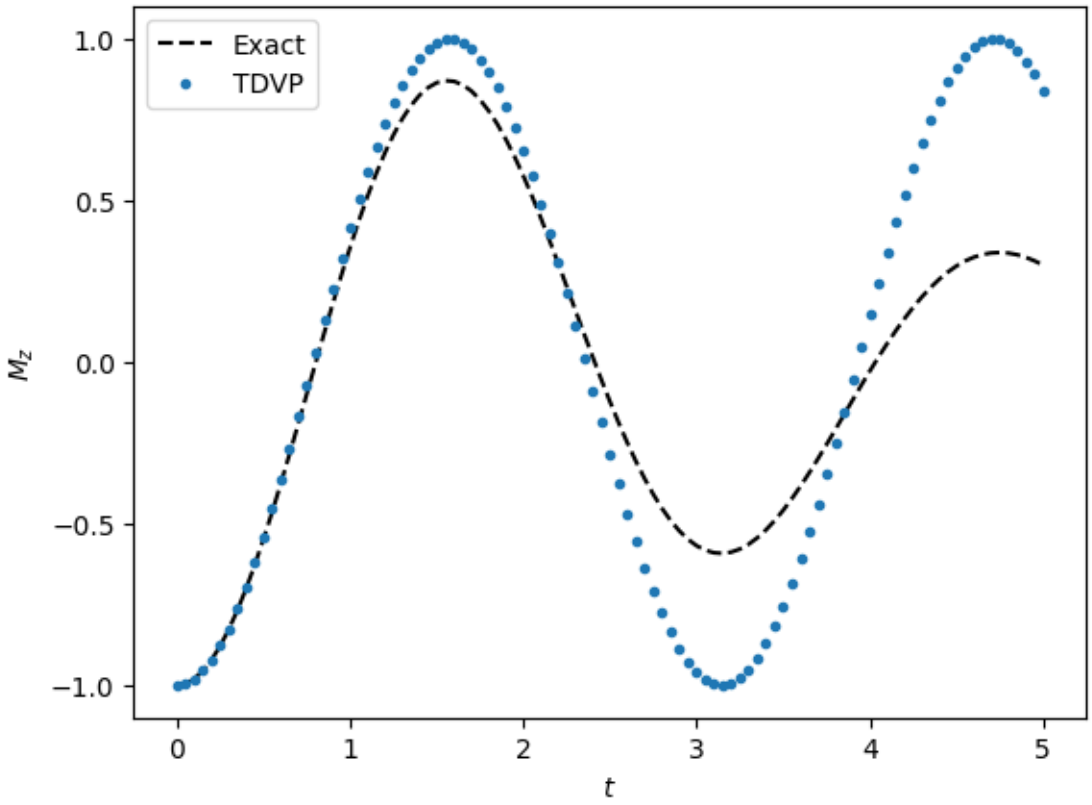
[30]:
```
# TDVP

# Create the list of operators to be measured
sx_list = list_of_op(sx, N)
sz_list = list_of_op(sz, N)

sxsx_list = []
for i in range(N):
    sxsx_row = []
    for j in range(N):
        sxsx_row.append(sx_list[i] @ sx_list[j])
    sxsx_list.append(sxsx_row)

sx_hami_list = []
for sx_i in sx_list:
    sx_hami_list.append(sx_i @ hami)
```

[31]:
```
# Initialise the TDVP loop

 _params = np.zeros(N)

# Initial state
spin_up = np.array([1, 0])
```

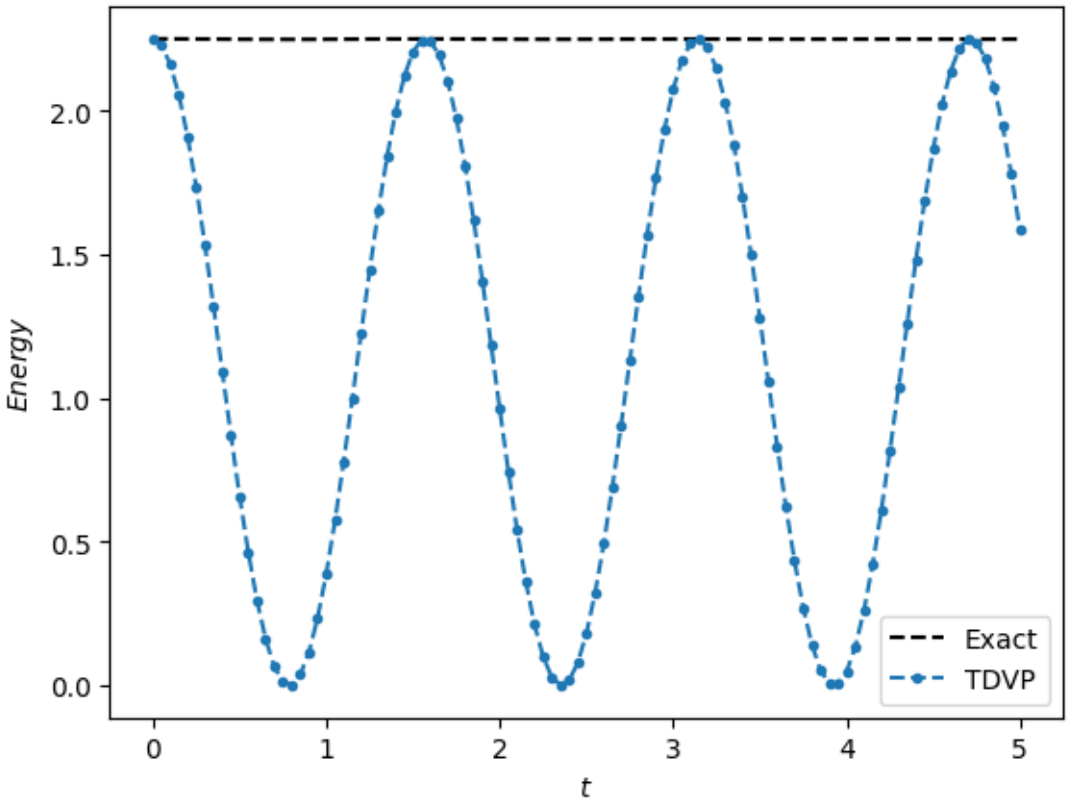
```
spin_down = np.array([0, 1])

psi0 = 1
for i in range(N):
    psi0 = np.kron(psi0, spin_down)

# And initial variational ansatz
psi_op = mf_ansatz_op(N,  _params)
psi_var = mf_ansatz_psi(psi_op, psi0)
psi_norm = squared_norm(psi_var)

mz_op = Mz(N)
mz_tdvp = []
mz_tdvp.append(measure(psi0, mz_op))


# Energy
e_tdvp = []
e_tdvp.append(measure(psi0, hami))
```

[32]:
```
for step in range(nt):
    # 1) Measure all the operators to evaluate S and C
    sx_meas = np.asarray([measure(psi_var, x) for x in sx_list])
    sxsx_meas = np.asarray(
        [[measure(psi_var, x) for x in sxsx_row] for sxsx_row in sxsx_list]
    )
    energy = measure(psi_var, hami)
    sx_hami_meas = np.asarray([measure(psi_var, x) for x in sx_hami_list])

    # 2) Create S and C
    S_mat = S( _params, sx_meas, sxsx_meas, psi_norm)
    C_vec = C( _params, energy, sx_meas, sx_hami_meas, psi_norm)

    # 3) Solve the linear system
     _dot = solve(np.real(S_mat), np.imag(C_vec))

    # 4) Update the parameters
     _params =  _params +  _dot * dt

    # 5) Create the new variational state and measure observables
    psi_op = mf_ansatz_op(N,  _params)
    psi_var = mf_ansatz_psi(psi_op, psi0)
    psi_norm = squared_norm(psi_var)

    mz_tdvp.append(measure(psi_var, mz_op))
    e_tdvp.append(measure(psi_var, hami))
```

[33]:
```
# Plot and compare
plt.plot(ts, mz_t, linestyle="dashed", color="black", label="Exact")
plt.plot(ts, mz_tdvp, linestyle="", color="C0", marker=".", label="TDVP")
plt.ylabel(r"$M_z$")
plt.xlabel(r"$t$")
plt.yticks([-1, -0.5, 0, 0.5, 1])
plt.legend()
plt.show()
```



[34]:
```
# Now check the energy
plt.plot(ts, e_t, linestyle="dashed", color="black", label="Exact")
plt.plot(ts, e_tdvp, linestyle="dashed", color="C0", marker=".", label="TDVP")
plt.ylabel(r"$Energy$")
plt.xlabel(r"$t$")
plt.legend()
plt.show()
```



0.1.2 Problem 8.2 Imaginary time evolution

In this exercise we will use the imaginary time evolution method to find the ground state of an Ising spin chain and periodic boundary conditions.

[35]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy.sparse import csr_array, identity, kron
from scipy.sparse.linalg import eigsh
```

[36]:
```
# Pauli matrices
sx = csr_array([[0, 1], [1, 0]])
sz = csr_array([[1, 0], [0, -1]])
```

a) TFIM Hamiltonian as in Problem 3.1

[37]:
```
def operator(pauli, i, N):
    left = identity(2**i)
    right = identity(2 ** (N - i - 1))
    mat = kron(kron(left, pauli), right)

    # Explicitly convert to CSR since kron may return COO
```

```
    mat = csr_array(mat)
    return mat
```

[38]:
```
def gen_hamiltonian(sx_list, sz_list, J, Gamma, pbc):
    N = len(sx_list)
    ham = 0
    for i in range(N - 1 + pbc):
        ham -= Gamma * sx_list[i]
        j = (i + 1) % N
        ham += J * sz_list[i] @ sz_list[j]
    return ham
```

b) Exact diagonalization as in Problem 3.1

[39]:
```
N = 10
J = 1
Gammas = [0.5, 1, 1.5, 2]
sx_list = [operator(sx, i, N) for i in range(N)]
sz_list = [operator(sz, i, N) for i in range(N)]

Hs = []
energies_0 = []
energies_1 = []
gaps = []
for Gamma in Gammas:
    H = gen_hamiltonian(sx_list, sz_list, J, Gamma, pbc=True)

    # If we set `k=2`, the Lanczos algorithm may not really converge to the two␣
 ↪smallest eigenvectors
    # From my experience it's safe to set `k` to the double of what we need
    exact = eigsh(H, k=4, which="SA", return_eigenvectors=False)

    Hs.append(H)
    energies_0.append(exact[-1])
    energies_1.append(exact[-2])
    gaps.append(exact[-2] - exact[-1])

print("E_0:", energies_0)
print("Gaps:", gaps)
```

```
E_0: [np.float64(-10.635604409347945), np.float64(-12.784906442999315),
np.float64(-16.7230249139484), np.float64(-21.271208818695968)]
Gaps: [np.float64(0.0003207266806573017), np.float64(0.15740341364924326),
np.float64(1.0075698441114795), np.float64(2.0006414533614176)]
```

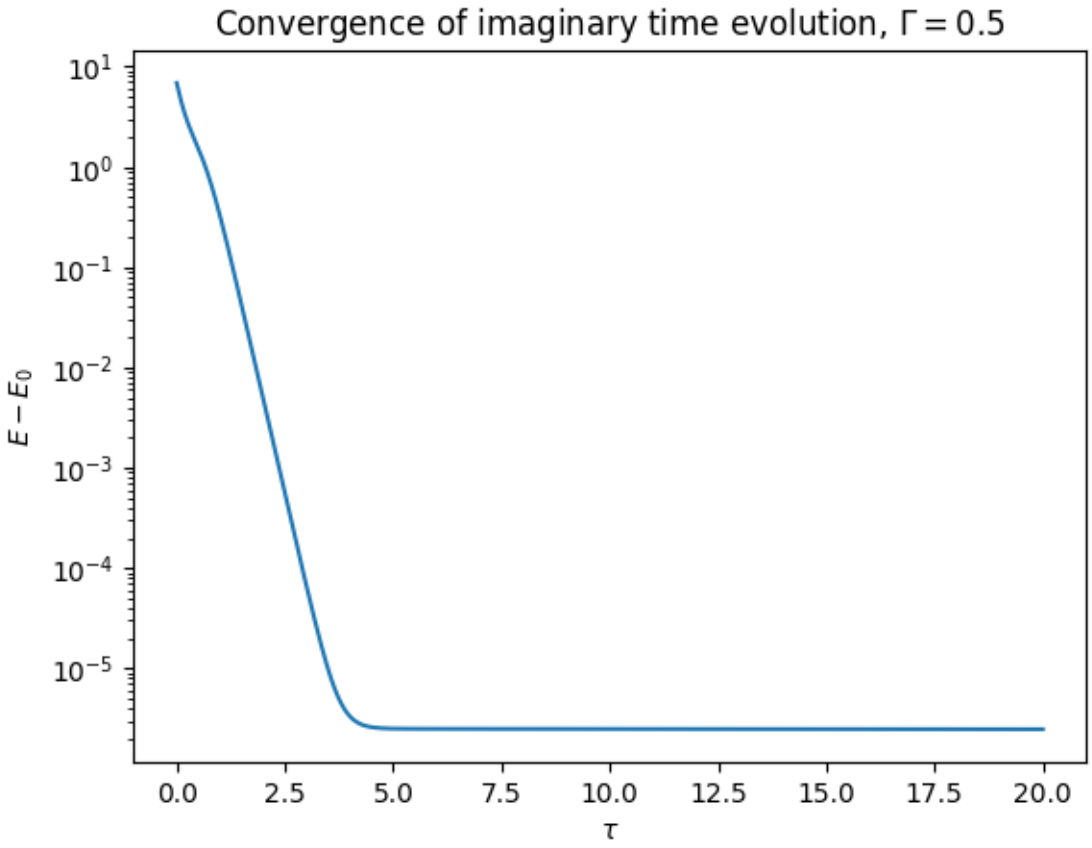
c) Implement imaginary time evolution

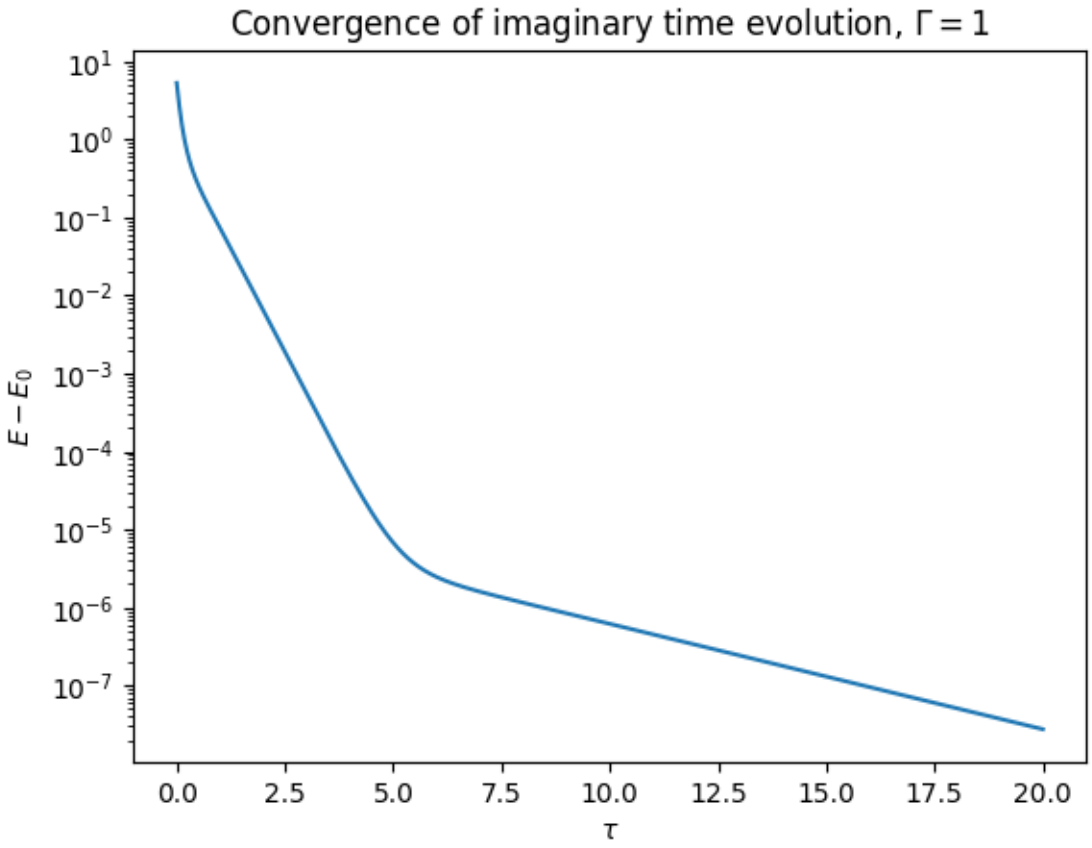
[40]:
```
def imag_time_evol(H, psi, tau, dtau):
    n_steps = round(tau / dtau)
    for step in range(n_steps):
        psi -= dtau * (H @ psi)
        psi /= np.sqrt(psi.conj() @ psi)
    return psi
```

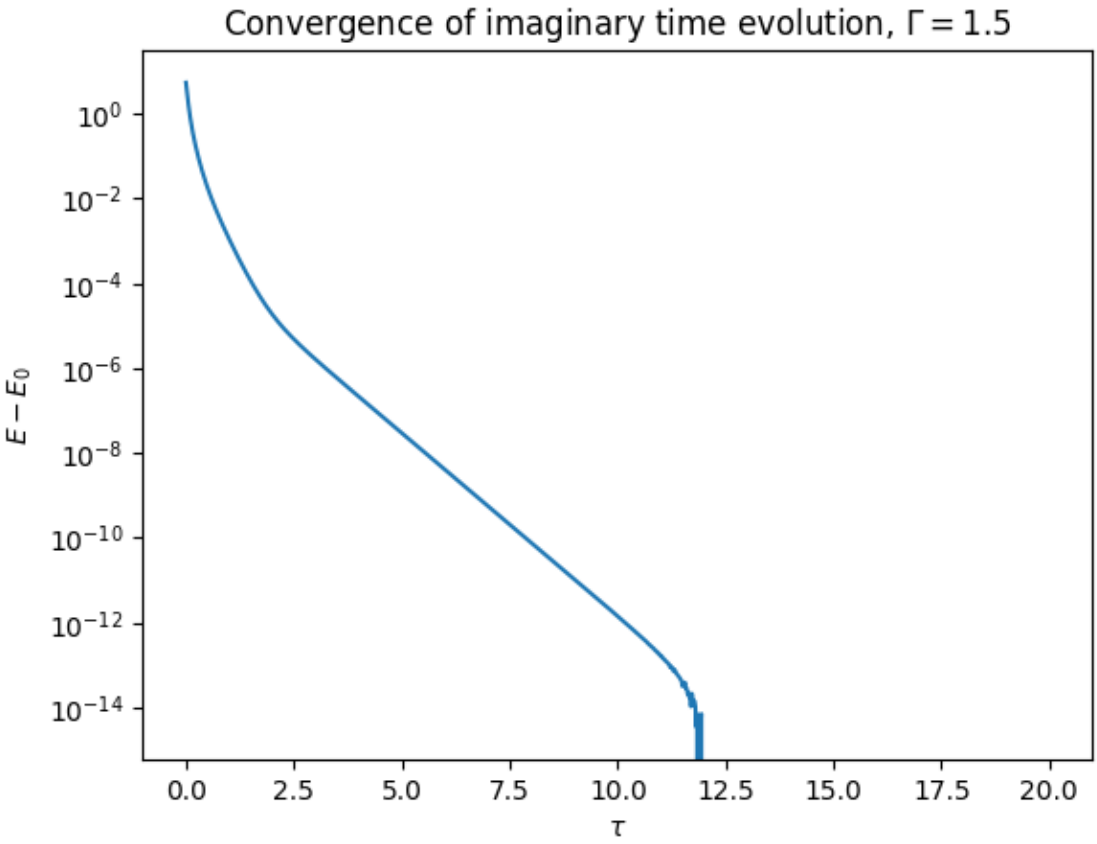
[41]:
```
def measure(H, psi):
    return (psi.conj() @ (H @ psi)).real


dtau = 1e-3
n_points = 1000
tau = 20

taus = np.linspace(0, tau, n_points + 1)
energies_ite = np.empty((len(Gammas), n_points + 1))

for k in range(len(Gammas)):
    H = Hs[k]

    # Random initial psi
    psi = np.random.rand(2**N) + np.random.rand(2**N) * 1j
    psi /= np.sqrt(psi.conj() @ psi)

    energies_ite[k, 0] = measure(H, psi)
    for i in range(1, n_points + 1):
        psi = imag_time_evol(H, psi, tau / n_points, dtau)
        energies_ite[k, i] = measure(H, psi)
```

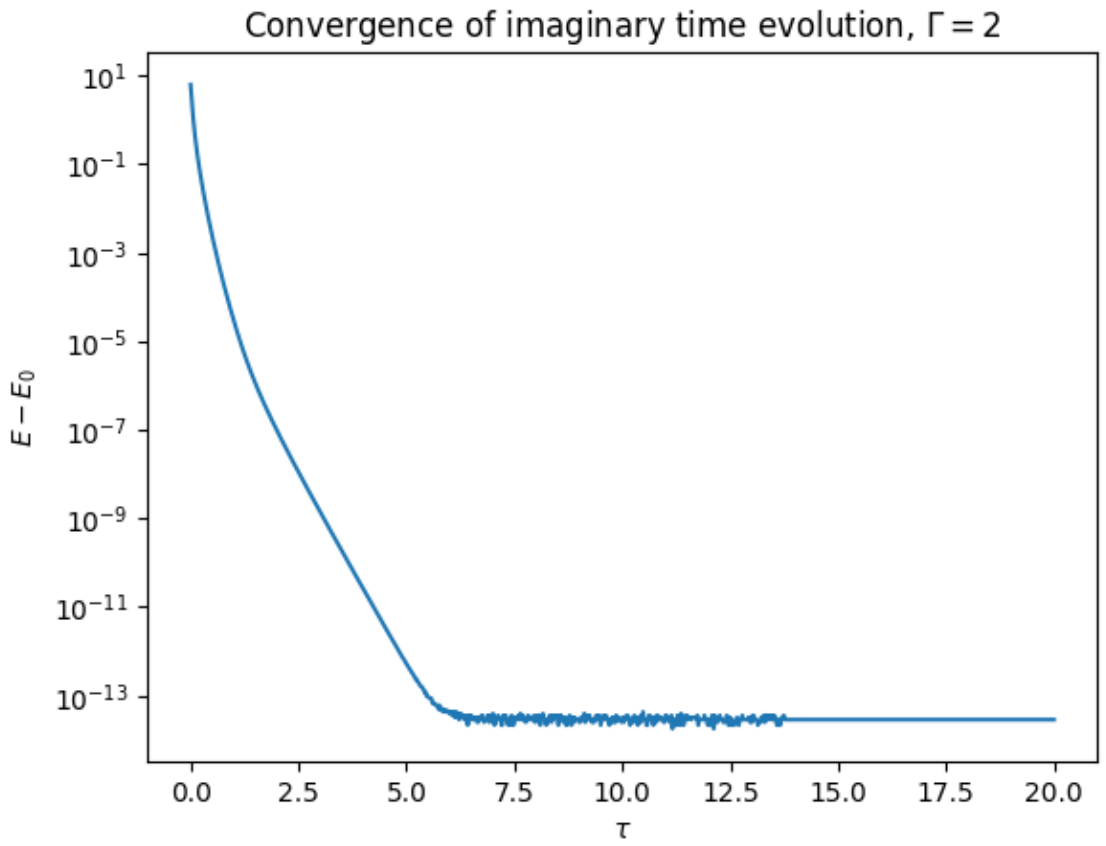
d) Plot the energies

[42]:
```
for k in range(len(Gammas)):
    plt.plot(taus, energies_ite[k] - energies_0[k])
    plt.xlabel("$\\tau$")
    plt.ylabel("$E - E_0$")
    plt.yscale("log")
    plt.title(f"Convergence of imaginary time evolution, $\\Gamma =␣
 ↪{Gammas[k]}$")
    plt.show()
```


Convergence of imaginary time evolution, Γ = 0.5
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
$E - E_0$
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
τ


Convergence of imaginary time evolution, Γ = 1
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
$E - E_0$
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
τ


Convergence of imaginary time evolution, $\Gamma = 1.5$
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
$E - E_0$
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
τ

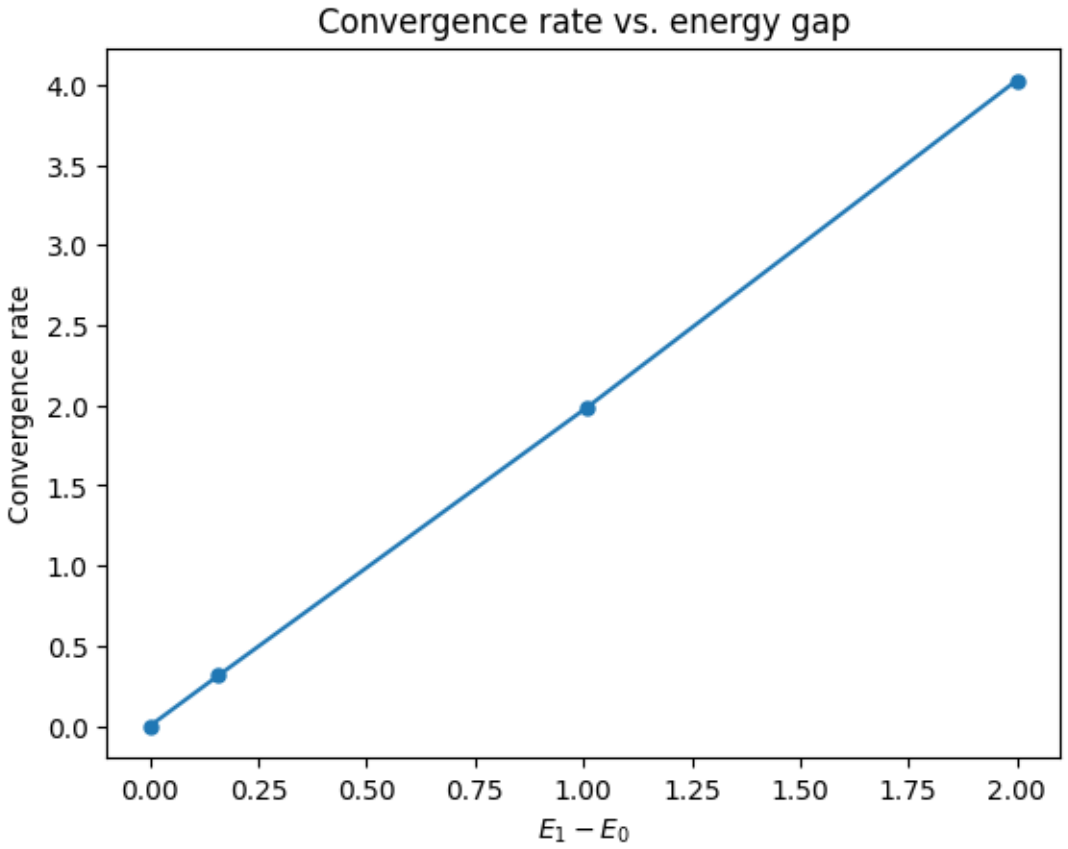


e) Fit the convergence rates

[43]:
```
intervals = [(6, 18), (6, 18), (4, 10), (2, 5)]

rates = []
for k in range(len(Gammas)):
    gap = gaps[k]

    mask = (intervals[k][0] < taus) & (taus < intervals[k][1])
    a = np.polyfit(taus[mask], np.log(energies_ite[k, mask] - energies_0[k]), 1)
    rate = -a[0]
    rates.append(rate)

    print(gap, rate)
```

```
0.0003207266806573017 0.00063026212304085
0.15740341364924326 0.31340464088635817
1.0075698441114795 1.9847204857923895
2.0006414533614176 4.0271936035324565
```

[44]:
```
plt.plot(gaps, rates, marker=".", markersize=10)
plt.xlabel("$E_1 - E_0$")
plt.ylabel("Convergence rate")
plt.title("Convergence rate vs. energy gap")
plt.show()
```



When τ is large enough and the components of $|\psi_2\rangle$ and higher excited states are decayed negligible, but $|\psi_1\rangle$ is still significant, we have

$$|\psi(\tau)\rangle = \mathrm{e}^{-E_0\tau}\left(c_0|\psi_0\rangle + \mathrm{e}^{-\Delta E\tau}c_1|\psi_1\rangle\right), \tag{1}$$

where $\Delta E = E_1 - E_0$. The evolved energy is

$$E(\tau) = \frac{c_0^2 E_0 + \mathrm{e}^{-2\Delta E\tau}c_1^2 E_1}{c_0^2 + \mathrm{e}^{-2\Delta E\tau}c_1^2} = E_0 + \Delta E\lambda + O(\lambda^2), \tag{2}$$

where $\lambda = \left(\frac{c_1}{c_0}\mathrm{e}^{-\Delta E\tau}\right)^2$. When fitting the exponential decay, we can find

$$\log\left(E(\tau) - E_0\right) = -2\Delta E\tau + \log\left(\Delta E\frac{c_1^2}{c_0^2}\right), \tag{3}$$

so the convergence rate is linear in ΔE.