ex03_solution

March 11, 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 03 - Quantum spin models

0.1.1 Problem 3.1 Exact diagonalization of the transverse field Ising Hamiltonian

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

a) Start by defining a function to generate the operator

$$\hat{\sigma}_i^\alpha = \underbrace{\hat{I} \otimes \hat{I} \otimes \ldots \hat{I}}_{i-1 \text{ times}} \otimes \hat{\sigma}^\alpha \otimes \underbrace{\hat{I} \otimes \cdots \otimes \hat{I}}_{N-i \text{ times}})$$

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

[3]:
```
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 likes to return COO
    mat = csr_array(mat)
    return mat
```

[4]:
```
op = operator(sx, 0, 2)
print(type(op))
```

```
<class 'scipy.sparse._csr.csr_array'>
```

b) Using sparse matrices for the pauli operators $\hat{\sigma}_i^\alpha$ defined above we write a function which constructs the transverse field ising hamiltonian

$$\hat{H}_{\text{Ising}} = J \sum_{i=1}^{N} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z - \Gamma \sum_{i=1}^{N} \hat{\sigma}_i^x$$

[5]:
```
def tfi_hamiltonian(N, J, gamma):
    H = 0
    for i in range(N):
        j = (i+1)%N
        H += J * operator(sz, i, N) @ operator(sz, j, N)
    for i in range(N):
        H -= gamma * operator(sx, i, N)
    return H
```

We can check that for 2 spins we get the matrix given on the exercise sheet:

[6]:
```
n_spins = 2
J= 1
gamma = 0.1

print(tfi_hamiltonian(n_spins, J, gamma).todense())
```

```
[[ 2.  -0.1 -0.1  0. ]
 [-0.1 -2.   0.  -0.1]
 [-0.1  0.  -2.  -0.1]
 [ 0.  -0.1 -0.1  2. ]]
```

c) Use scipy to diagonalize the hamiltonian

[7]:
```
n_spins = 10

H = tfi_hamiltonian(n_spins, J, gamma)

# 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
E_ed = eigsh(H, k=4, which="SA", return_eigenvectors=False)
print(E_ed)
```

```
[ -6.38306387  -6.40332833 -10.02501566 -10.02501566]
```

Implement the power method

[8]:
```
# lambda is a reserved keyword for lambda functions,
# so we can't give the variable like that name....
def power_method(H, n_spins, n_iters, lambd=1, u0=None):
    if u0 is None:
        # take a random vector as in initial guess
        # it might need to be complex
```

```
        u = np.random.normal(size=2**n_spins) + 0j
    else:
        u = u0

    # construct the propagator
    I = csr_array(identity(2**n_spins))
    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

        old = e
        energies.append(e)

    return u, energies
```

For the TFI in 1D with periodic boundary conditions we can upper bound the largest eigenvalue E_M with

$$E_M \leq |J|N + |\Gamma|N$$

and so to satisfy $\Lambda > E_M$ we can choose

$$\Lambda = |J|N + |\Gamma|N$$

Note that it is also possible to derive a tighter condition for Λ than given in the lecture notes *(see Becca & Sorella 2017, pp. 34)*, namely $\Lambda > \frac{E_0+E_M}{2}$. This means that for the TFI with even N the power method should converge for any $\Lambda > 0$.

[9]:
```
lambd = abs(J)*n_spins + abs(gamma)*n_spins

v, E_pwr = power_method(H, n_spins, n_iters=2000, lambd=lambd)
E_pwr[-1]
```

```
/var/folders/79/jcbbyf357tn_ctl5c6wsbn1h0000gp/T/ipykernel_92695/1040166075.py:3
2: RuntimeWarning: invalid value encountered in scalar divide
  if np.abs((old - e) / e) < 1e-15:
```

[9]:
```
(-10.025015664202867+0j)
```

and check that both find the same ground state energy

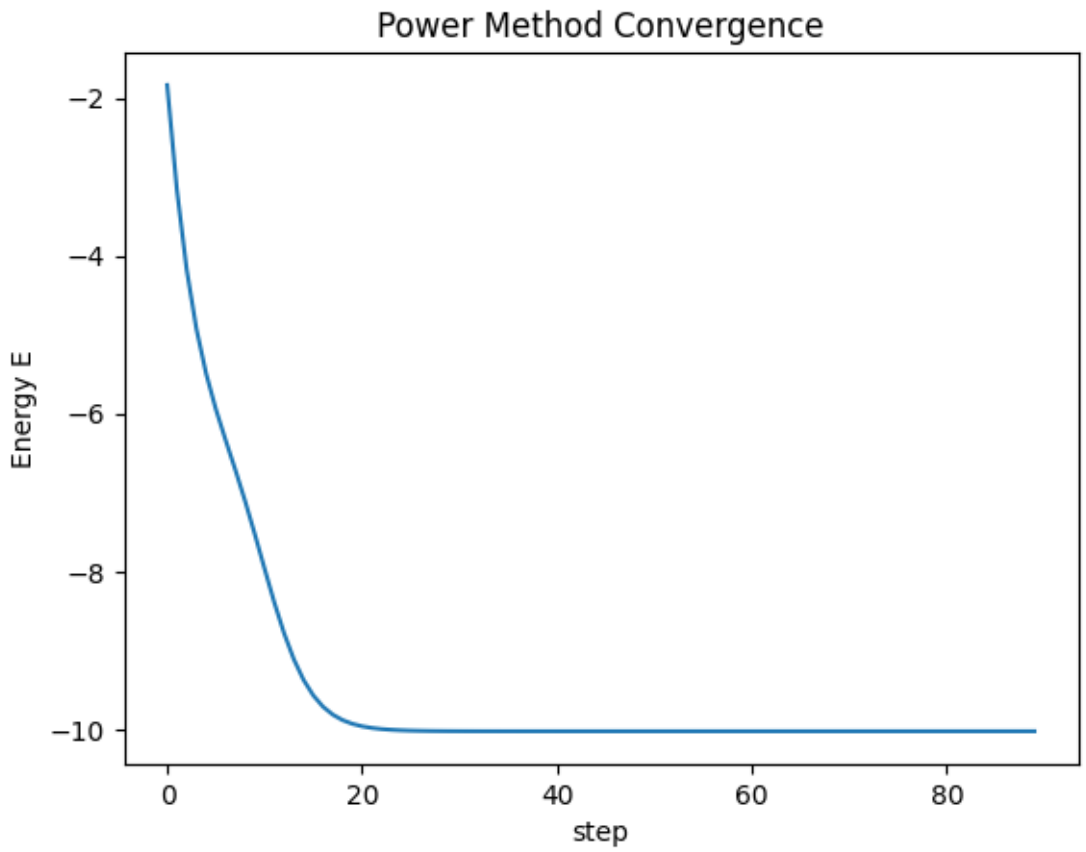
[10]:
```
np.allclose(E_pwr[-1], E_ed[-1])
```

[10]:
```
True
```

We see that the power method converges nicely already after a few steps

[11]:
```
plt.figure()
plt.plot(E_pwr)
plt.ylabel("Energy E")
plt.xlabel("step")
plt.title('Power Method Convergence')
plt.show()
```

```
/Users/alessandrosinibaldi/Desktop/Coding/PhD/.venv/lib/python3.11/site-
packages/matplotlib/cbook.py:1762: ComplexWarning: Casting complex values to
real discards the imaginary part
  return math.isfinite(val)
/Users/alessandrosinibaldi/Desktop/Coding/PhD/.venv/lib/python3.11/site-
packages/matplotlib/cbook.py:1398: ComplexWarning: Casting complex values to
real discards the imaginary part
  return np.asarray(x, float)
```



d) Proceed by computing the ground state energy for different system sizes N and transverse field strengths Γ and studying the spin-spin correlator

$$C = \langle \psi_0 | \hat{\sigma}_0^z \hat{\sigma}_{N/2}^z | \psi_0 \rangle$$

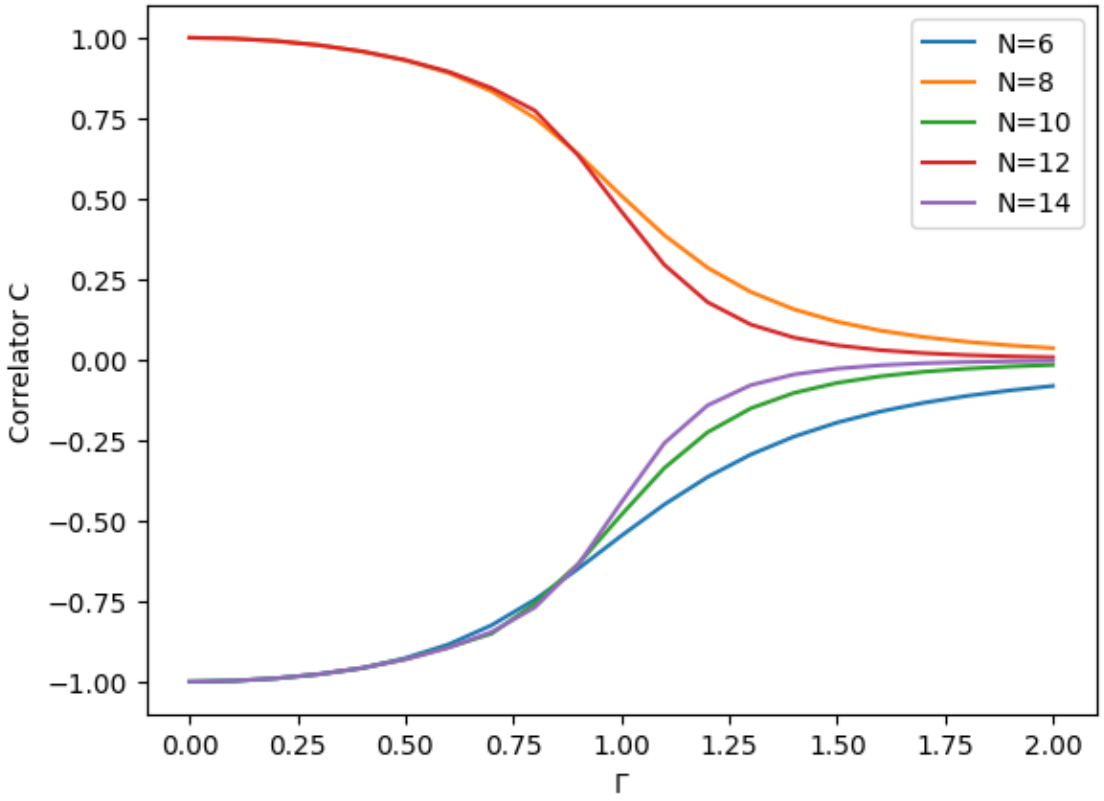
[12]:
```
Ns = [6, 8, 10, 12, 14]
gs = np.linspace(0, 2, 21)

corr = np.zeros((len(Ns), len(gs)))

plt.figure()
for i, N in enumerate(Ns):
    print(f"N={N}")
    for j, g in enumerate(gs):
        H = tfi_hamiltonian(N, J=1, gamma=g)
        v, e = power_method(H, N, 1000)
        corr[i, j] = v.T.conj() @ (operator(sz, 0, N) @ (operator(sz, N//2, N)␣
 ↪@ v))
    plt.plot(gs, corr[i, :], label=f"N={N}")
plt.ylabel("Correlator C")
```

```
plt.xlabel("$\Gamma$")
plt.legend()
plt.show()
```

```
N=6
```

```
/var/folders/79/jcbbyf357tn_ctl5c6wsbn1h0000gp/T/ipykernel_92695/1040166075.py:3
2: RuntimeWarning: invalid value encountered in scalar divide
  if np.abs((old - e) / e) < 1e-15:
/var/folders/79/jcbbyf357tn_ctl5c6wsbn1h0000gp/T/ipykernel_92695/2552400123.py:1
2: ComplexWarning: Casting complex values to real discards the imaginary part
  corr[i, j] = v.T.conj() @ (operator(sz, 0, N) @ (operator(sz, N//2, N) @ v))
```

```
N=8
N=10
N=12
N=14
```

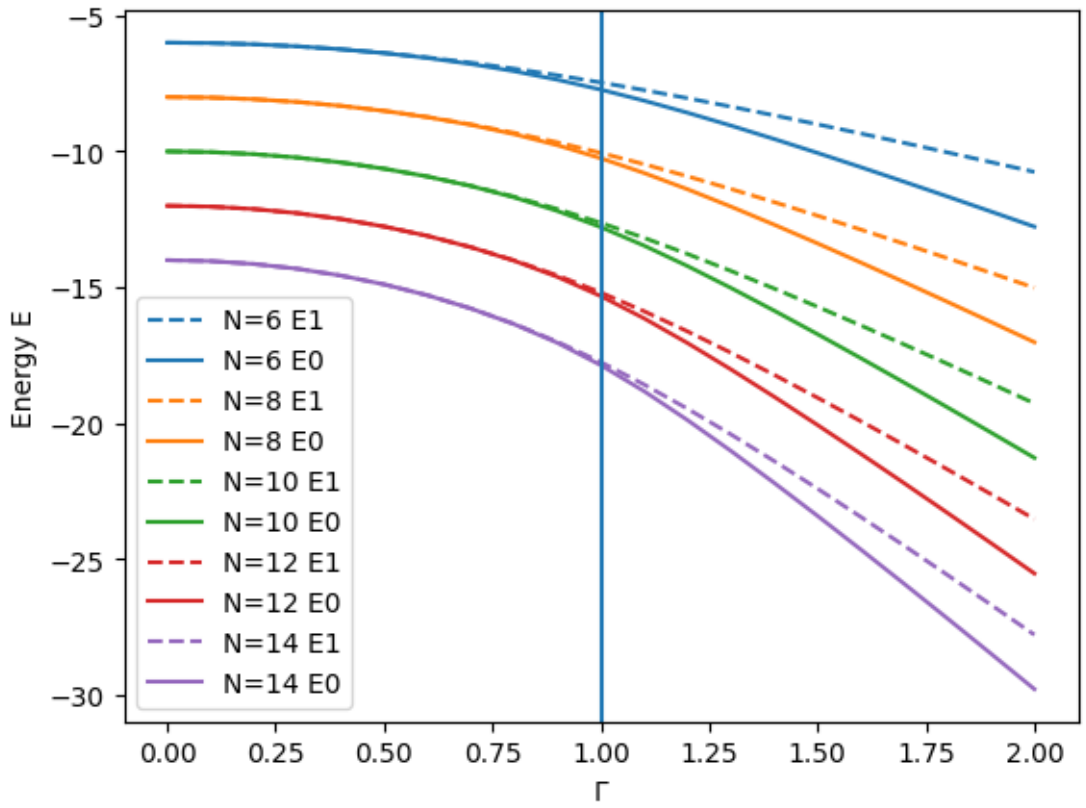


e) Compute the the ground state as well as first excited state energy using ED and plot both

[13]:
```
Ns = [6, 8, 10, 12, 14]
gs = np.linspace(0, 2, 21)
```

```
k = 4  # how many eigenvalues to compute

energies = np.zeros((len(Ns), len(gs), k))

for i, N in enumerate(Ns):
    for j, g in enumerate(gs):
        H = tfi_hamiltonian(N, J=1, gamma=g)
        # we sort the energies in increasing order,
        # so that energies[:, :, 0] is the ground state energy
        energies[i, j, :] = np.sort(eigsh(H, k=k, which="SA",␣
 ↪return_eigenvectors=False))
```

[14]:
```
plt.figure()
for i, N in enumerate(Ns):
    plt.plot(gs, energies[i, :, 1], label=f"N={N} E1", color=f"C{i}",␣
 ↪linestyle="--")
    plt.plot(gs, energies[i, :, 0], label=f"N={N} E0", color=f"C{i}")

plt.axvline((1.0))
plt.ylabel("Energy E")
plt.xlabel("$\Gamma$")
plt.legend()
plt.show()
```



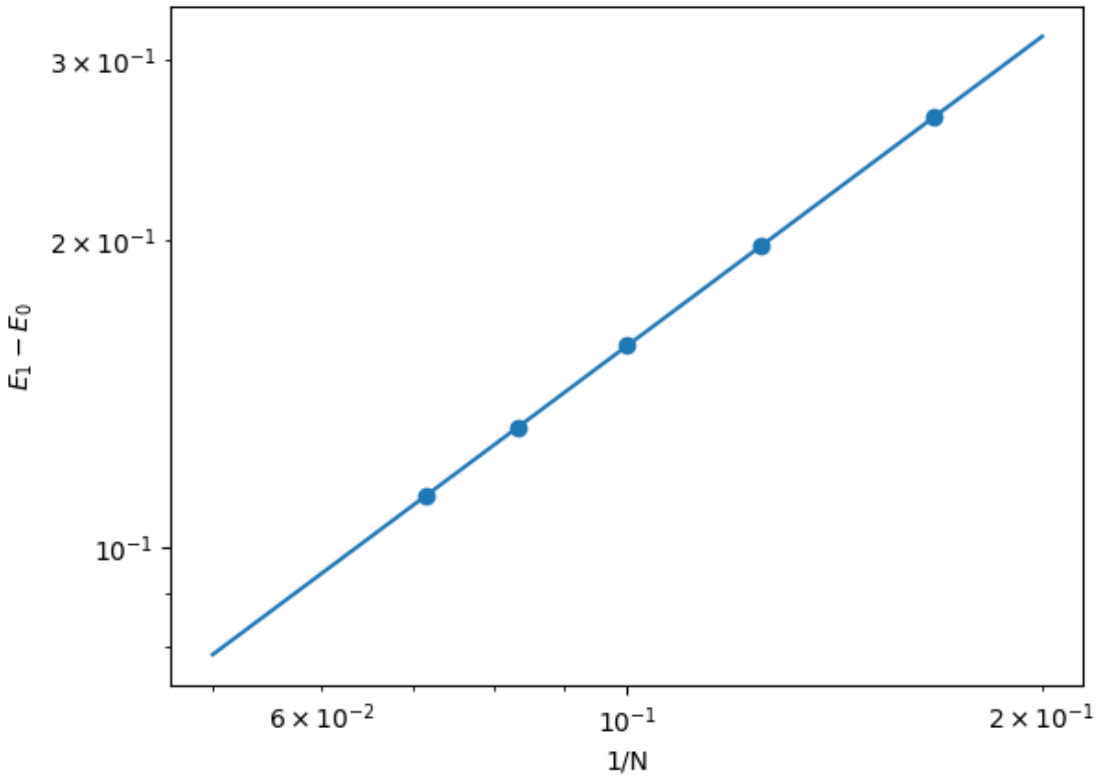
and plot the gap at the critical point $\Gamma/J = 1$ and extrapolate to $N \to \infty$ (thermodynamic limit)

[15]:
```
i = len(gs)//2
print(f'Γ = {gs[i]}')
gap = energies[:, i, 1] - energies[:, i, 0]

Ninv = 1 / np.array(Ns)
# fit
a = np.polyfit(np.log(Ninv), np.log(gap), 1)
f = lambda x: np.exp(a[0]*np.log(x)+a[1])

print(f'slope: {a[0]:0.5f}')
print(f"Energy gap at 1/N -> 0: {f(1/np.inf):0.5f}")

plt.figure()
plt.scatter(Ninv, gap)
x = np.linspace(0.05, 0.2, 100)
plt.plot(x, f(x))
plt.xscale('log')
plt.yscale('log')
```

```
plt.xlabel('1/N')
plt.ylabel('$E_1-E_0$')
plt.show()
```

```
Γ = 1.0
slope: 1.00548
Energy gap at 1/N -> 0: 0.00000
```

```
/var/folders/79/jcbbyf357tn_ctl5c6wsbn1h0000gp/T/ipykernel_92695/1676215558.py:8
: RuntimeWarning: divide by zero encountered in log
  f = lambda x: np.exp(a[0]*np.log(x)+a[1])
```



0.1.2 Problem 3.2 Time evolution of the transverse field Ising chain

[16]:
```
import numpy as np
from matplotlib import pyplot as plt
from scipy.sparse import csr_array, identity, kron
from functools import lru_cache
```

a) We define the time evolution operator for the transverse field part

$$\exp(-i\delta\hat{\sigma}_i^x)$$

[17]:
```python
@lru_cache # helps save computation time
def exp_x(N, i, delta):

    I = identity(2**N)
    Xi = operator(sx, i, N)

    mat = np.cos(delta) * I - 1j * np.sin(delta) * Xi
##else
    # TODO mat = ...
    mat = csr_array(mat)
    return mat
```

b) as well as the Time evolution operator for the diagonal part

$$\exp(-i\delta\hat{\sigma}_i^z\sigma_{i+1}^z)$$

[18]:
```python
@lru_cache
def exp_zz(N, i, delta):

    Zi = operator(sz, i, N)
    # here we write the code for pbc,
    # for obc we assume i only goes to N-2
    Zi_p_1 = operator(sz, (i+1)%N, N)
    I = identity(2**N)

    mat = np.cos(delta) * I - 1j * np.sin(delta) * Zi@Zi_p_1
    mat = csr_array(mat)
    return mat
```

c) Putting everything together define a function which performs the time evolution

$$\exp(-i\Delta_t\hat{H}) \approx \exp(-i\frac{\Delta_t}{2}J\hat{H}_{\mathrm{ZZ}})\exp(-i\Delta_t\Gamma\hat{H}_{\mathrm{X}})\exp(-i\frac{\Delta_t}{2}J\hat{H}_{\mathrm{ZZ}}) + O(\Delta_t^3)$$

[19]:
```python
def time_evol(psi, t, dt, J, gamma):
    n_steps = round(t / dt)

    for i in range(N-1): # N-1 because we assume obc
        psi = exp_zz(N, i, J*dt/2) @ psi

    for i in range(N):
        psi = exp_x(N, i, gamma*dt) @ psi

    for step in range(n_steps):

        for i in range(N-1):
            # here we fuse the two exp_Hzz(N, i, dt/2) of two
```

```
            # consecutive steps into one exp_Hzz(N, i, dt)
            psi = exp_zz(N, i, J*dt) @ psi

        for i in range(N):
            psi = exp_x(N, i, gamma*dt) @ psi

    for i in range(N-1):
        psi = exp_zz(N, i, J*dt/2) @ psi
    return psi
```

d) Study the time evolution of a small system of size 10

[20]:
```
N = 10
```

To measure the Magnetisation of each site define the operator $\langle \sigma_i^z \rangle$

[21]:
```
@lru_cache # to save computation time
def Hz(N, i):
    mat = operator(sz, i, N)
    mat = csr_array(mat)
    return mat
```

Starting with an initial state with all-down configuration

$$|\Psi\rangle = |\downarrow\rangle \otimes |\downarrow\rangle \otimes ... \otimes |\downarrow\rangle$$

[22]:
```
# `psi` can be dense, and we can store `psi` as a 1D vector rather than a 2D␣
 ↪matrix

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

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

run the time evolution for J=0, $\Gamma = 1$

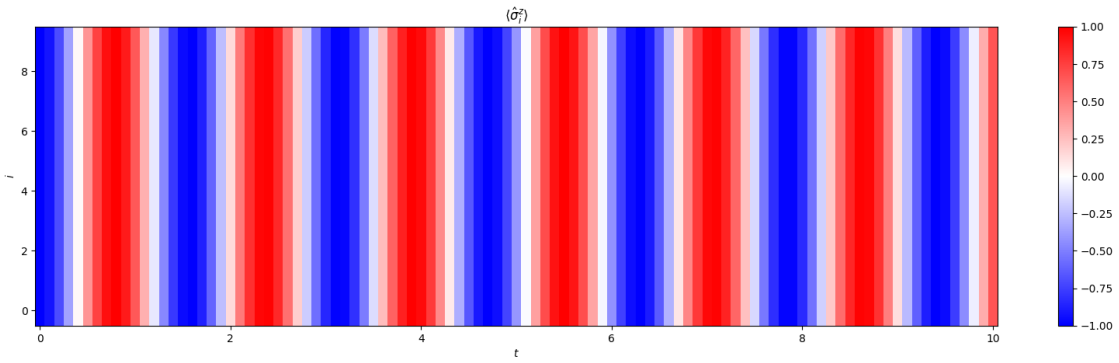
[23]:
```
J = 0
gamma = 1

dt = 1e-1
t = 10
n_points = 100
ts = np.linspace(0, t, n_points+1)

# we run the time evolution up to t=10
# with a timestep of dt=1e-2,
# measuring the observables every t/100 = 0.1
```

```
psi_t = psi.copy()

def measure(psi, observalbes):
    return [((psi.conj().T @ O) @ psi) for O in observalbes]

observables = [Hz(N, i) for i in range(N)]
magnetization = []
###Ifdef SOLUTION
magnetization.append(measure(psi, observables))
for i in range(1, n_points+1):
    psi_t = time_evol(psi_t, t/n_points, dt, J, gamma)
    magnetization.append(measure(psi_t, observables))
```

e) and plot the magnetization $\langle \sigma_i^z \rangle$ over time for each site i

[24]:
```
magnetization = np.array(magnetization).real.T
```

[25]:
```
plt.figure(figsize=(20, 5))
plt.pcolormesh(ts, range(N), magnetization, cmap="bwr", vmin=-1, vmax=1,␣
↪shading="nearest")
plt.xlabel("$t$")
plt.ylabel("$i$")
plt.title("$\\langle \\hat \\sigma^z_i \\rangle$")
plt.colorbar()
plt.show()
```



f) Do the same as in e) but for an initial state with all spins down except one spin up

$$|\Psi\rangle = |\downarrow\rangle \otimes ... \otimes |\uparrow\rangle \otimes ... \otimes |\downarrow\rangle$$

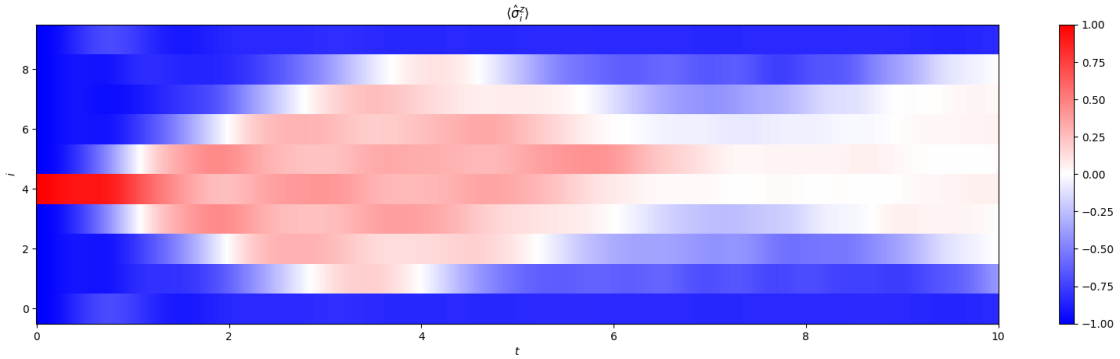
and J=1, $\Gamma = 0.4$

[26]:
```
# Initial state
psi = 1
for i in range((N - 1) // 2):
    psi = np.kron(psi, spin_down)
psi = np.kron(psi, spin_up)
for i in range(N // 2):
    psi = np.kron(psi, spin_down)
```

[ ]:
```
# Time evolution
J = 1
gamma = 0.4

# this time we run for longer time
dt = 1e-2
t = 10
n_points = 1000
ts = np.linspace(0, t, n_points+1)

psi_t = psi.copy()

def measure(psi, observalbes):
    return [((psi.conj().T @ O) @ psi) for O in observalbes]

observables = [Hz(N, i) for i in range(N)]
magnetization = []
###Ifdef SOLUTION
magnetization.append(measure(psi, observables))
for i in range(1, n_points+1):
    psi_t = time_evol(psi_t, t/n_points, dt, J, gamma)
    magnetization.append(measure(psi_t, observables))
```

[28]:
```
magnetization = np.array(magnetization).real.T
```

[29]:
```
plt.figure(figsize=(20, 5))
plt.pcolormesh(ts, range(N), magnetization, cmap="bwr", vmin=-1, vmax=1,␣
 ↪shading="nearest")
plt.xlabel("$t$")
plt.ylabel("$i$")
plt.title("$\\langle \\hat \\sigma^z_i \\rangle$")
plt.colorbar()
plt.show()
```


$\langle \tilde{\sigma}_i^z \rangle$
i
t
0
2
4
6
8
10
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00