Project 2

Quantum Mechanics

December 12, 2022

C Seitz

Part 1

Finite Differences Method

For the first part of the project, I use the finite difference in the time domain (FDTD) method to solve Schrodinger's equation. The algorithm is well-known, so I will just sketch the major result I use here. The idea is the break up the time-dependent Schrodinger equation into coupled differential equations for the real and imaginary parts of the wavefunction.

Let ℓ be a discrete spatial coordinate and m, n index time for the real $\psi_R(\ell)$ and imaginary $\psi_I(\ell)$ parts of the wavefunction, respectively. The update equations are

$$\frac{1}{\Delta\tau}\left(\psi_I^{n+1}(\ell) - \psi_I^n(\ell)\right) = \eta\left(\psi_R^m(\ell+1) - 2\psi_R^m(\ell) + \psi_R^m(\ell-1)\right) - V(\ell)\psi_R^m(\ell)$$

$$\frac{1}{\Delta\tau}\left(\psi_R^{m+1}(\ell) - \psi_R^m(\ell)\right) = -\eta\left(\psi_I^n(\ell+1) - 2\psi_I^n(\ell) + \psi_I^n(\ell-1)\right) - V(\ell)\psi_I^n(\ell)$$

where $\tau = t/\hbar$ and $\eta = \frac{\hbar^2}{2m\Delta x^2}$ (and we set $\eta = 1$) is the hopping parameter. You solve the system numerically by alternatively updating the real and imaginary wavefunctions in time. Suppose the initial state is $|\alpha\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$ where $|0\rangle$ and $|1\rangle$ are the ground state and first excited state for the infinite square well. Then we find the position representation $\langle i|0\rangle$ and $\langle i|1\rangle$ by solving the time-independent Schrodinger equation as in Project 1. We then can construct $\langle i|\alpha\rangle$, which is purely real, and assign this as the initial condition $\psi_R^0(\ell)$.

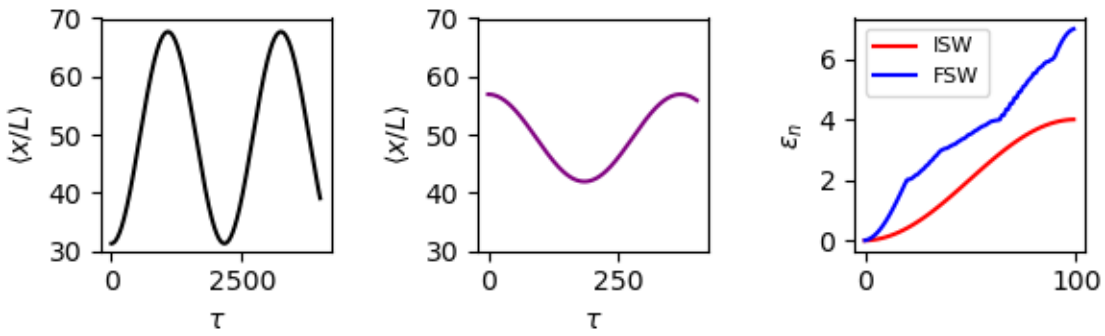

Figure 1: Expectation values of position as a function of time for the infinite (left) and finite (middle) square well, plus the eigenvalue spectrum for both (right). $d\tau = 0.01, \eta = 1, N_x = 100, N_t = 4 \times 10^5$

Answers

(A) Since the wavefunction is a superposition of eigenkets, I expect the probability density to change in time, probably in an oscillatory fashion.
(B) The time scale should depend on the energies of the eigenkets of the superposition
(C) See Figure 1 for FDTD simulation of the infinite and finite square wells
(D) For any superposition state, we can write

$$\langle x \rangle = \sum_{a'} \sum_{a''} c^*_{a'} c_{a''} \langle a' | x | a'' \rangle \exp\left(\frac{-i(E_{a''} - E_{a'})t}{\hbar}\right)$$

which in this particular case (using $\tau = t/\hbar$) is

$$\langle x \rangle = \frac{1}{2}\left(\langle 0 | x | 1 \rangle \exp\left(i\Delta E \tau\right) + \langle 1 | x | 0 \rangle \exp\left(-i\Delta E \tau\right)\right)$$

The angular frequency will be higher for the finite square well because $\Delta E = E_1 - E_0$ is larger see the derivative of the eigenvalue spectrum in Figure 1).
(E) See email attachments for the gif animation corresponding to the simulations in Figure 1

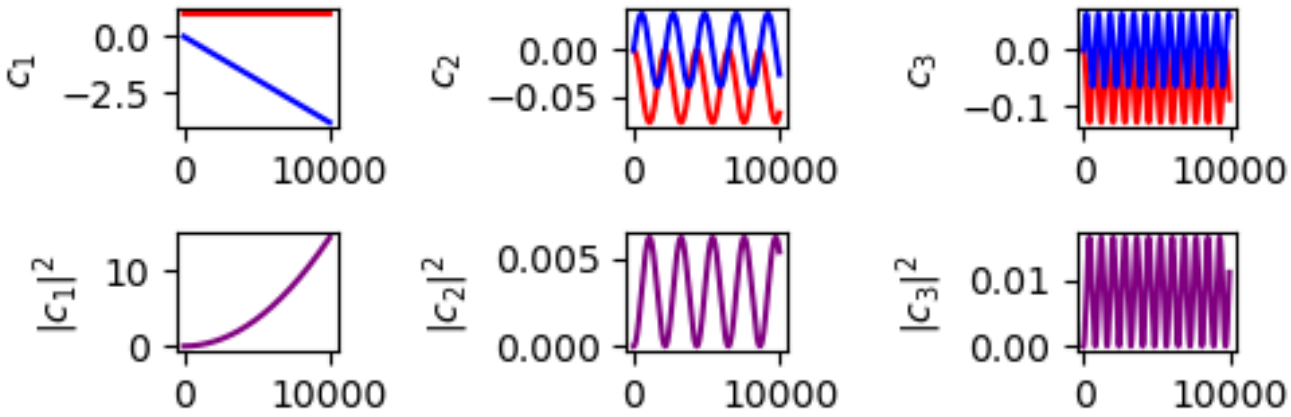

Figure 2: Left column: First order $c_1(t)$, transition probability $P(1 \to 1) = |c_1^{(1)}(t)|^2$. Middle column: $c_2(t)$ and $P(1 \to 2) = |c_2^{(1)}(t)|^2$. Right column: $c_3(t)$ and $P(1 \to 3) = |c_3^{(1)}(t)|^2$. $d\tau = 0.1, N_t = 1 \times 10^5$

Part 2

Time-dependent peturbation theory for a constant perturbation

We are using the time-dependent Hamiltonian

$$H(x,t) = H_0(x) + \lambda(1 - e^{-t/\tau})V(x)$$

where $V(x)$ is the finite square well potential. We are assuming that $\tau \to \infty$ so the potentially turns on exactly at $t = 0$, giving a **constant perturbation**.

Answers

(F) If $\lambda = 0$, we do not expect $\langle x/L \rangle$ to oscillate because $H(t) = H_0$ and $|\psi\rangle = |i\rangle$ which is an eigenstate of H_0.
(G) If $\lambda \neq 0$, we do expect $\langle x/L \rangle$ to oscillate because turning on the potential $V(x)$ makes $|\psi\rangle$ a superposition of eigenkets of the $H(t) = H_0 + \lambda V(x)$
(H)

Now, we are told that we start in the ground state of H_0, which is the Hamiltonian for an infinite square well, i.e. $|\psi(0\rangle = |i\rangle = |1\rangle$. recall that in time-dependent first order perturbation theory, Schrodinger's equation becomes

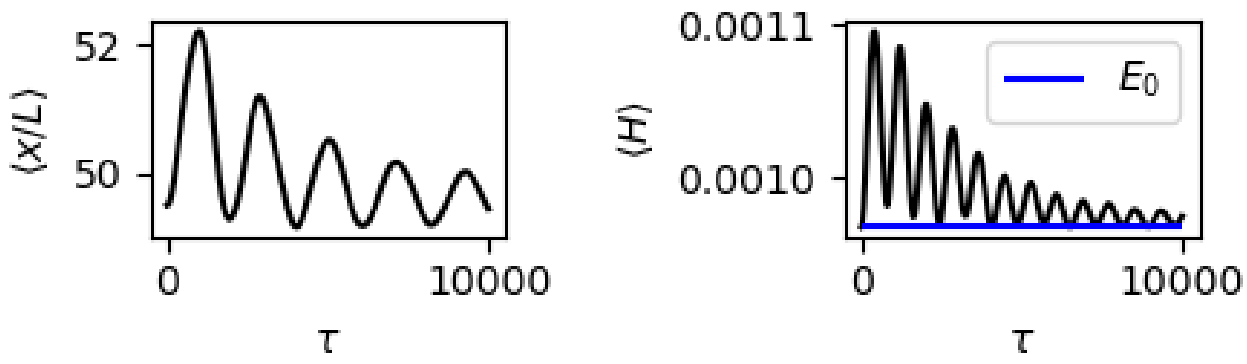

Figure 3: Expectation value of position (left) and energy (right) after turning on the perturbation

$$
\begin{aligned}
i\hbar\frac{d}{dt}c_n^{(0)}(t) &= 0 \\
i\hbar\frac{d}{dt}c_n^{(1)}(t) &= \sum_i V_{ni}e^{i\omega_{ni}t}c_i^{(0)}(t)
\end{aligned}
$$

where $V_{ni} = \langle n|\, V\, |i\rangle$ and $c_n(t)$ is the expansion coefficient for **unperturbed** energy eigenket $|n\rangle$. In general, expansion coefficients for energy eigenkets are, to first order

$$
\begin{aligned}
c_n(t) &\approx c_n^{(0)}(t) + \lambda c_n^{(1)}(t) \\
&= \delta_{ni} - \frac{i\lambda}{\hbar}\int_0^t V_{ni}e^{i\omega_{ni}t}dt
\end{aligned}
$$

To first order, the transition probabilities $P(i \to n) = |c_n^{(1)}(t)|^2$. From the expression above, we have

$$
c_n^{(1)}(t) = -\frac{i}{\hbar}\int_0^t V_{ni}e^{i\omega_{ni}t}dt
$$

For states with $E_n \neq E_i$,

$$
c_n(t) \approx \delta_{ni} + \frac{V_{ni}}{E_n - E_i}(1 - e^{i\omega_{ni}t})
$$

and

$$P(i \to n) = \frac{4|V_{ni}|^2}{|E_n - E_i|^2} \sin^2\left(\frac{(E_n - E_i)t}{2\hbar}\right) \tag{1}$$

and for states with $E_n = E_i$,

$$c_n(t) \approx \delta_{ni} - \frac{iV_{ni}}{\hbar}t$$

and

$$P(i \to n) = |V_{ni}|^2 t^2 \tag{2}$$

To summarize, the transition probability oscillates but the amplitude of this oscillation is small provided the magnitude of the transition matrix element V_{ni} of the perturbation is small compared to the unperturbed energy difference between the initial and final states. However, if $E_n = E_i$, the transition probability grows quadratically in time

(I) To compute the wavefunction $\langle i|\alpha(\tau)\rangle$ we need both the magnitude and phase of the first order corrections to the expansion coefficients. Using those, we can directly construct the superposition:

$$\langle i|\alpha(\tau)\rangle = c_1(t)\psi_1(\ell) + c_2(t)\psi_2(\ell) + c_3(t)\psi_3(\ell)$$

The equations to find $c_n(t)$ were given in part (H). We ensure that $\langle i|\alpha(\tau)\rangle$ is normalized for all τ by dividing the wavefunction by $\sqrt{\sum_i \langle i|\alpha(\tau)\rangle}$

(J) I wasn't able to get this part to work correctly with my FDTD method, so my animation is generated by computing the coefficients from part (I) directly and constructing the superposition:

$$\langle i|\alpha(\tau)\rangle = \frac{c_1(t)\psi_1(\ell) + c_2(t)\psi_2(\ell) + c_3(t)\psi_3(\ell)}{\sqrt{A}}$$

for $A = \sum_i |\langle i|\alpha\rangle|^2$.

(K) As $\tau \to \infty$ we should have the wavefunction approach that of the ground state, because as you can see in Figure 2, the transition probability $P(1 \to 1)$ grows quadratically in time while the other transition probabilities oscillate sinusoidally.

(L) The expectation value $\langle x/L \rangle$ is plotted as a function of time in Figure 3. When the potential is turned on, the particle goes from a stationary state to a superposition and then slowly back to a stationary state for the same reason as in (K).
(M) The expectation value $\langle H \rangle$ is plotted as a function of time in Figure 3.
(N) The true energy, $H_0 + V$ is conserved. It is only because we are considering the dynamics in terms of the unperturbed system, using the unperturbed energy defined by H_0, that we can speak of energy non-conservation from the unperturbed point of view
(O) At long times $\langle H \rangle \to E_0$

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
from plt2array import plt2array
from skimage.io import imsave

class FDTDSolver:
    def
        __init__(self,Nx,Nt,V,psi_r0,psi_i0,dir,plot=True,plot_iter_num=10,dt=0.1,name='sim',H=
        self.Nt = Nt
        self.Nx = Nx
        self.V = V
        self.psi_r0 = psi_r0
        self.psi_i0 = psi_i0
        self.psi_r = psi_r0
        self.psi_i = psi_i0
        self.psi_r = np.pad(self.psi_r, (1,1))
        self.psi_i = np.pad(self.psi_i, (1,1))
        self.prob = np.zeros((Nx,Nt))
        self.prob[:,0] = self.psi_r[1:-1]**2 + self.psi_i[1:-1]**2
        self.c1 = dt
        self.c2 = dt
        self.dir = dir
        self.plot_iter_num = plot_iter_num
        self.plot = plot
        self.name = name
        self.dt = dt
        self.H = H
        self.Etau = np.zeros((Nt,))

    def compute_energy(self,t):
        psi = self.psi_r[1:-1] + self.psi_i[1:-1]*1j
        self.Etau[t] = np.conjugate(psi) @ self.H @ psi

    def plot_init_(self):
        fig, ax = plt.subplots(figsize=(3,1))
        ax.plot(self.psi_r0,color='red')
        ax.plot(self.psi_i0,color='blue')
        plt.tight_layout()
        plt.show()

    def plot_iter(self,t):
        fig, ax = plt.subplots()
        ax1 = ax.twinx()
        ax.set_ylim([-0.1,0.1])
        ax1.set_ylim([-2*self.V.max(),2*self.V.max()])
        ax.plot(self.psi_r,color='red',label=r'$\psi_{R}$')
        ax.plot(self.psi_i,color='blue',label=r'$\psi_{I}$')
```

```
        ax.plot(self.psi_i**2 +
            self.psi_r**2,color='purple',label=r'$|\psi|^{2}$')
        ax1.plot(self.V[:,t],color='black',label=r'$V(x)$')
        ax1.set_ylabel('V(x)')
        ax.text(0.05, 0.95, r'$\tau$' + f'={self.dt*t}', transform=ax.transAxes,
            fontsize=14,
        verticalalignment='top')
        ax.legend()
        plt.tight_layout()
        rgb_array = plt2array(fig)
        imsave(self.dir+f'{t}_{self.name}.tif',rgb_array)
        plt.close()

    def update_r(self,t):
        for n in range(1,self.Nx+1):
            self.psi_r[n] = self.psi_r[n]-\
            self.c1*(self.psi_i[n+1] - 2*self.psi_i[n] + self.psi_i[n-1]) +\
            self.c2*self.V[n,t]*self.psi_i[n]

    def update_i(self,t):
        for n in range(1,self.Nx+1):
            self.psi_i[n] = self.psi_i[n] + \
                   self.c1*(self.psi_r[n+1] - 2*self.psi_r[n] + self.psi_r[n-1])\
                   - self.c2*self.V[n,t]*self.psi_r[n]
    def forward(self):
        if self.plot:
            self.plot_iter(0)
        for t in range(1,self.Nt):
            if self.H is not None:
                self.compute_energy(t)
            print(f'Time step: {t}')
            self.update_i(t)
            self.update_r(t)
            self.prob[:,t] = self.psi_r[1:-1]**2 + self.psi_i[1:-1]**2
            if t % self.plot_iter_num == 0:
                if self.plot:
                    self.plot_iter(t)
```

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
from fdtd import FDTDSolver

dir = '/home/cwseitz/Desktop/temp/'
Nx = 100
Nt = 400000
eta = 1
dtau = 0.01
```

```
######################
# Infinite square well
######################

print('Simulating the infinite square well...\n')
tau1 = np.arange(0,Nt)*dtau
V = np.zeros((Nx,Nt))
H = np.zeros((Nx,Nx)) #Hamiltonian at t = 0
H += np.diag(2*eta + V[:,0],k=0) #main diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=1) #upper diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=-1) #lower diagonal

##########################################
# Find eigenvectors and eigenvalues of H0
##########################################

vals, vecs = LA.eig(H)
idx = np.argsort(vals)
vecs1 = vecs[:,idx]
vals1 = vals[idx]

################################################
# Simulate time evolution in a infinite square well
################################################

V = np.pad(V, ((1,1),(0,0)))
psi_r0 = (vecs1[:,0] + vecs1[:,1])/np.sqrt(2)
psi_i0 = np.zeros_like(psi_r0)
solver1 =
    FDTDSolver(Nx,Nt,V,psi_r0,psi_i0,dir,plot_iter_num=10000,plot=True,dt=dtau,name='inf',H=H)
solver1.forward()

######################
# Finite square well
######################

Nt = 40000
print('Simulating the finite square well...\n')
tau2 = np.arange(0,Nt)*dtau
Vl = 2
Vr = 3
V = np.zeros((Nx,Nt))
V[:30,:] = Vl
V[70:,:] = Vr
H = np.zeros((Nx,Nx)) #Hamiltonian at t = 0
H += np.diag(2*eta + V[:,0],k=0) #main diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=1) #upper diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=-1) #lower diagonal
```

```
##########################################
# Find eigenvectors and eigenvalues of H0
##########################################

vals, vecs = LA.eig(H)
idx = np.argsort(vals)
vecs2 = vecs[:,idx]
vals2 = vals[idx]

################################################
# Simulate time evolution in a finite square well
################################################

V = np.pad(V, ((1,1),(0,0)))
psi_r0 = (vecs2[:,0] + vecs2[:,1])/np.sqrt(2)
psi_i0 = np.zeros_like(psi_r0)
solver2 =
    FDTDSolver(Nx,Nt,V,psi_r0,psi_i0,dir,plot_iter_num=1000,plot=True,dt=dtau,name='fin',H=H)
solver2.forward()

################################################
# Plot expectation values of scaled position
################################################

fig, ax = plt.subplots(1,3,figsize=(6,2))
X_avg = solver1.prob.T * np.arange(0,Nx)
X_avg = np.sum(X_avg,axis=1)
ax[0].plot(tau1,X_avg,color='black')
ax[0].set_xlabel(r'$\tau$')
ax[0].set_ylabel(r'$\langle x/L\rangle$')
ax[0].set_ylim([30,70])
X_avg = solver2.prob.T * np.arange(0,Nx)
X_avg = np.sum(X_avg,axis=1)
ax[1].plot(tau2,X_avg,color='purple')
ax[1].set_xlabel(r'$\tau$')
ax[1].set_ylabel(r'$\langle x/L\rangle$')
ax[1].set_ylim([30,70])
ax[2].plot(vals1,color='red',label='ISW')
ax[2].plot(vals2,color='blue',label='FSW')
ax[2].set_label('n')
ax[2].set_ylabel(r'$\epsilon_{n}$')
ax[2].legend(fontsize=8)
plt.tight_layout()
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from numpy import linalg as LA
from fdtd import FDTDSolver
from plt2array import plt2array
from skimage.io import imsave

dir = '/home/cwseitz/Desktop/temp/'
Nx = 100
Nt = 100000
eta = 1 #hopping parameter
dtau = 0.1

######################
# Infinite square well
######################

V = np.zeros((Nx,Nt))
H = np.zeros((Nx,Nx)) #Hamiltonian at t = 0
H += np.diag(2*eta + V[:,0],k=0) #main diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=1) #upper diagonal
H += np.diag(-eta*np.ones((Nx-1,)),k=-1) #lower diagonal

##########################################
# Find eigenvectors and eigenvalues of H0
##########################################

vals, vecs = LA.eig(H)
idx = np.argsort(vals)
vecs1 = vecs[:,idx]
vals1 = vals[idx]

#####################################
# Define the perturbation Hamiltonian
#####################################

Vl = 2
Vr = 3
V = np.zeros((Nx,Nt))
V[:30,:] = Vl
V[70:,:] = Vr
lam = 5 * 10**-4
V *= lam

##############################################
# Transform the perturbation to energy basis
##############################################

Hp = np.zeros((Nx,Nx)) #perturbation Hamiltonian
Hp += np.diag(V[:,0],k=0) #main diagonal
U0 = LA.inv(vecs1) #unitary operator
```

```
Hp_e = U0 @ Hp @ LA.inv(U0) #perturbation in energy basis
E0 = vals1[0]
E1 = vals1[1]
E2 = vals1[2]

##############################################
# Compute transition probability for n=1,2,3
##############################################

tau = np.arange(0,Nt)*dtau
v_11 = Hp_e[0,0]
v_12 = Hp_e[0,1]
v_13 = Hp_e[0,2]
c_1 = 1 - 1j*v_11*tau
omega_12 = vals1[1]-vals1[0]
c_2 = (v_12/omega_12)*(1-np.exp(1j*omega_12*tau))
omega_13 = vals1[2]-vals1[0]
c_3 = (v_13/omega_13)*(1-np.exp(1j*omega_13*tau))

fig, ax = plt.subplots(2,3,figsize=(9,2))
ax[0,0].plot(tau,np.real(c_1),color='red',label='Re')
ax[0,0].plot(tau,np.imag(c_1),color='blue',label='Im')
ax[0,0].set_ylabel(r'$c_{1}$')
ax1 = ax[1,0]
ax1.plot(tau, c_1*np.conj(c_1),color='purple')
ax1.set_ylabel(r'$|c_{1}|^{2}$')
ax[0,1].plot(tau,np.real(c_2),color='red',label='Re')
ax[0,1].plot(tau,np.imag(c_2),color='blue',label='Im')
ax[0,1].set_ylabel(r'$c_{2}$')
ax2 = ax[1,1]
ax2.plot(tau, c_2*np.conj(c_2),color='purple')
ax2.set_ylabel(r'$|c_{2}|^{2}$')
ax[0,2].plot(tau,np.real(c_3),color='red',label='Re')
ax[0,2].plot(tau,np.imag(c_3),color='blue',label='Im')
ax[0,2].set_ylabel(r'$c_{3}$')
ax3 = ax[1,2]
ax3.plot(tau, c_3*np.conj(c_3),color='purple')
ax3.set_ylabel(r'$|c_{3}|^{2}$')
plt.tight_layout()
plt.show()

########################################
# Construct time evolution analytically
########################################

probt = np.zeros((Nx,Nt))
probt[:,0] = vecs1[:,0]**2
Ht = np.zeros((Nt,))
Ht[0] = E0
```

```
for t in range(1,Nt):
    psi = c_1[t]*vecs1[:,0] + c_2[t]*vecs1[:,1] + c_3[t]*vecs1[:,2]
    prob = np.conj(psi)*psi
    A = np.sum(prob)
    prob = prob/A
    probt[:,t] = prob
    Z = c_1[t]*np.conj(c_1[t]) + c_2[t]*np.conj(c_2[t]) + c_3[t]*np.conj(c_3[t])
    Ht[t] = c_1[t]*np.conj(c_1[t])*E0 + c_2[t]*np.conj(c_2[t])*E1 +
        c_3[t]*np.conj(c_3[t])*E2
    Ht[t] = Ht[t]/Z
    if t % 1000 == 0:
        fig, ax = plt.subplots()
        ax.plot(prob)
        ax.set_ylim([0,0.2])
        rgb_array = plt2array(fig)
        imsave(dir+f'{t}_sim.tif',rgb_array)
        plt.close()

X_avg = probt.T * np.arange(0,Nx)
X_avg = np.sum(X_avg,axis=1)
fig, ax = plt.subplots(1,2)
ax[0].plot(tau,X_avg,color='black')
ax[0].set_xlabel(r'$\tau$')
ax[0].set_ylabel(r'$\langle x/L\rangle$')
ax[1].plot(tau,Ht,color='black')
ax[1].set_xlabel(r'$\tau$')
ax[1].set_ylabel(r'$\langle H \rangle$')
ax[1].hlines(E0,xmin=0,xmax=tau.max(),color='blue',label=r'$E_{0}$')
plt.legend()
plt.tight_layout()
plt.show()
```