

ex02_solution

February 25, 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 02 - Schroedinger equation in 1D

0.1.1 Problem 2.1 - Exact diagonalization for an Harmonic Well

```
[1]: ## import the libraries we will use for the exercise
import numpy as np
import scipy
from scipy.linalg import eigvalsh_tridiagonal, eigh_tridiagonal
import matplotlib.pyplot as plt

# use latex for plots
plt.rcParams['text.usetex'] = True
```

First, we create the function that returns the potential for the harmonic well

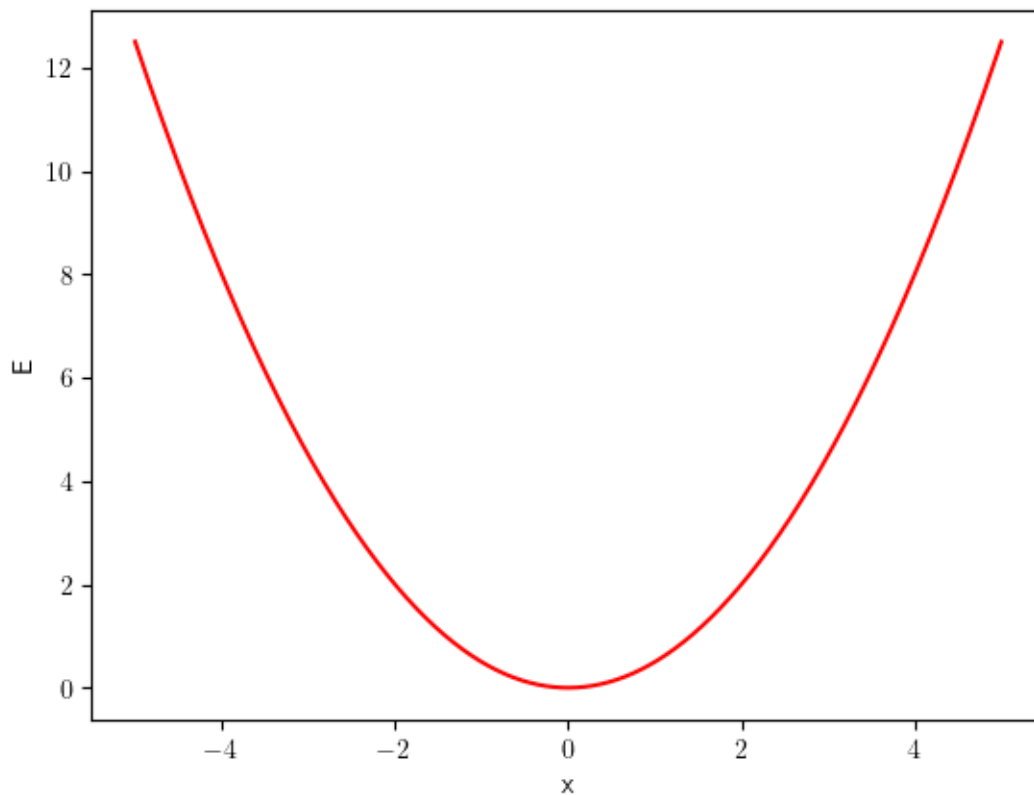
```
[2]: # this function is vectorised, i.e. it can act elementwise on a vector
def V(x):
    return 0.5*(x*x)
```

Next, we define the Hamiltonian as a matrix acting on the discretized 1D space we have chosen.

```
[3]: def hamiltonian(V, xs, dx, return_sparse_matrix=False):
    """
    compute the main and off-diagonal of the discretized hamiltonian for a 1D
    ↪ given potential V
    """
    diag = V(xs) + 1. / dx**2
    offdiag = -0.5 / dx**2 * np.ones(len(xs)-1)
    if return_sparse_matrix:
        return scipy.sparse.diags([diag, offdiag, offdiag], [0, -1, 1])
    else:
        return diag, offdiag
```

```
[4]: # define the interval and its discretization
xmin, xmax = -5,5
dx = 0.001
xs = np.arange(xmin, xmax, dx)
```

```
[5]: # plot the potential
plt.figure()
plt.plot(xs, V(xs), 'r')
plt.ylabel('E')
plt.xlabel('x')
plt.show()
```



The Harmonic Well has a well known exact solution in terms of Hermite polynomials. These functions are already present in SciPy.

```
[6]: from scipy.special import hermite
# Analytical solution for the harmonic oscillator
# assuming \hbar = m = 1
def psi(x,n):
    N = 1./np.sqrt(2**n * scipy.special.factorial(n)) * np.pi**(-0.25)
    return N * np.exp(-0.5*x**2) * hermite(n)(x)
```

```
def E(n):
    return n + 0.5
```

First, we define a function to obtain the eigenvalues and eigenvectors with $E < 5$ given the potential, the interval and the discretization.

```
[7]: def get_bound_states(V, xs, dx, n=1):
    main_diag, off_diag = hamiltonian(V, xs, dx)
    ## Here we select the eigenstates we want to keep, namely the first n
    E, psi = eigh_tridiagonal(main_diag, off_diag, select='i', select_range=(0,
↪n-1))
    return E, psi.T

def get_bound_eigvals(V, xs, dx, n=1):
    main_diag, off_diag = hamiltonian(V, xs, dx)
    ## Here we select the eigenstates we want to keep, namely the first n
    l = eigvalsh_tridiagonal(main_diag, off_diag, select='i', select_range=(0,
↪n-1))
    return l
```

Influence of discretization interval Now, we can see what happens to the energy levels if we vary the discretization interval dx , keeping fixed the interval $[x_{min}, x_{max}]$

```
[8]: ## Create a vector with different dx
dx_s = 2.**(-np.arange(1,13))
```

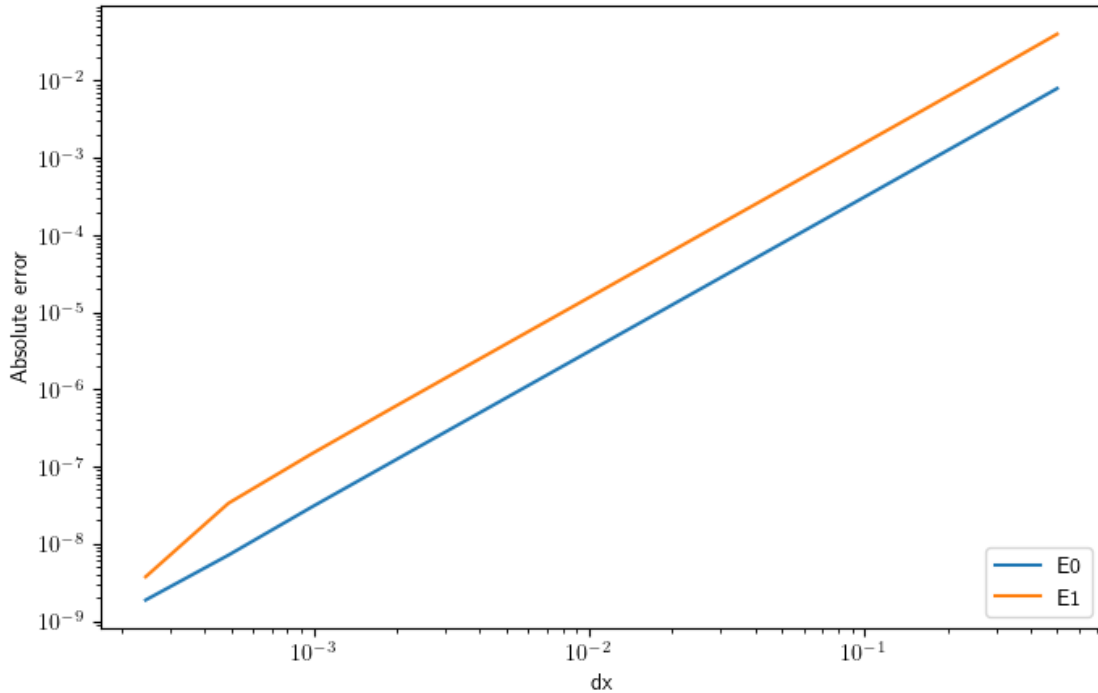
```
[9]: def plot_convergence_dx(V, xmin, xmax, dx_s):

    E_comp = []
    for dx in dx_s:
        xs = np.arange(xmin, xmax, dx)
        l = get_bound_eigvals(V, xs, dx, n=2)
        E_comp.append(l)

    E_comp = np.asarray(E_comp)

    plt.figure()
    plt.plot(dx_s, np.abs(E_comp[:,0]-E(0)), label="E0")
    plt.plot(dx_s, np.abs(E_comp[:,1]-E(1)), label="E1")
    plt.xlabel('dx')
    plt.ylabel('Absolute error')
    plt.xscale("log")
    plt.yscale("log")
    plt.legend(loc='lower right')
    plt.gcf().set_size_inches(8, 5)
    plt.show()
```

```
[10]: plot_convergence_dx(V, xmin, xmax, dx_s)
```



Influence of interval Now we vary the interval $[x_{min}, x_{max}]$, but keeping the density of points fixed. This means that the bigger the integral will be, the bigger the Hamiltonian matrix will get.

In particular, we define a vector of element x_{bound} and define a symmetric interval $[-x_{bound}, x_{bound}]$

```
[11]: x_bounds = np.linspace(2,100,100)
```

```
[12]: def plot_convergence_interval(V,x_bounds,dx):

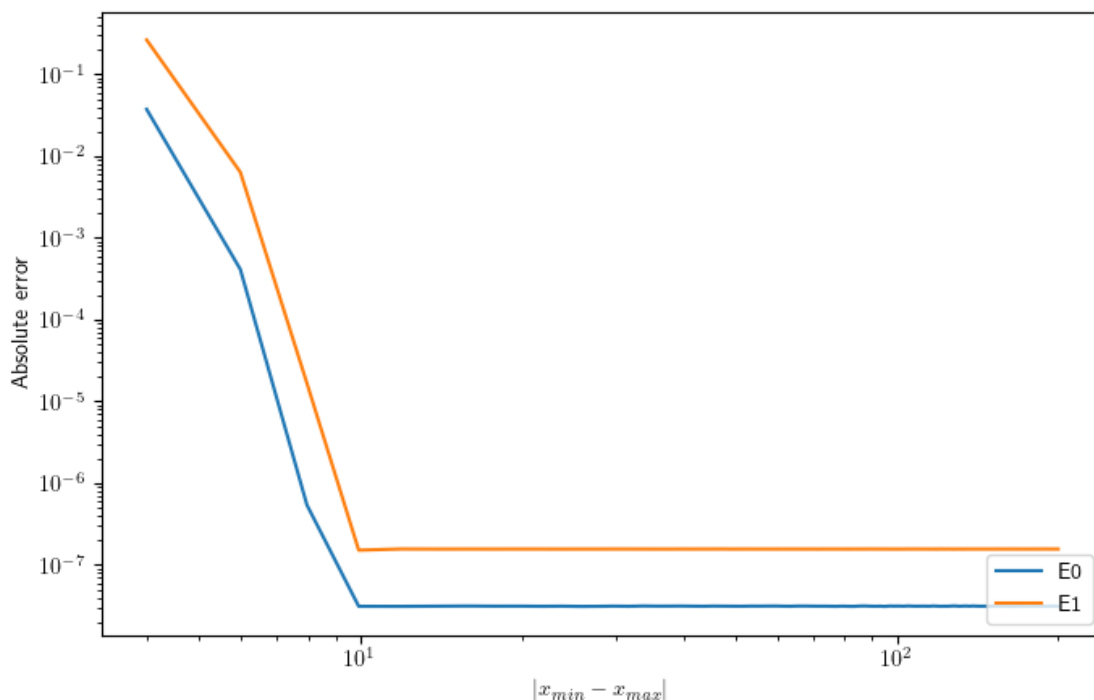
    E_comp = []
    for x0 in x_bounds:
        xs = np.arange(-x0, x0, dx)
        l = get_bound_eigvals(V, xs, dx,n=2)
        E_comp.append(l)

    E_comp = np.asarray(E_comp)

    plt.figure()
    plt.plot(2*x_bounds,np.abs(E_comp[:,0]-E(0)),label="E0")
    plt.plot(2*x_bounds,np.abs(E_comp[:,1]-E(1)),label="E1")
    plt.xlabel('$|x_{min} - x_{max}|$')
    plt.ylabel('Absolute error')
```

```
plt.xscale("log")
plt.yscale("log")
plt.legend(loc='lower right')
plt.gcf().set_size_inches(8, 5)
plt.show()
```

```
[13]: plot_convergence_interval(V,x_bounds,dx)
```



Bound states ($E < 5$) Finally, we plot all the eigenstates with $E < 5$

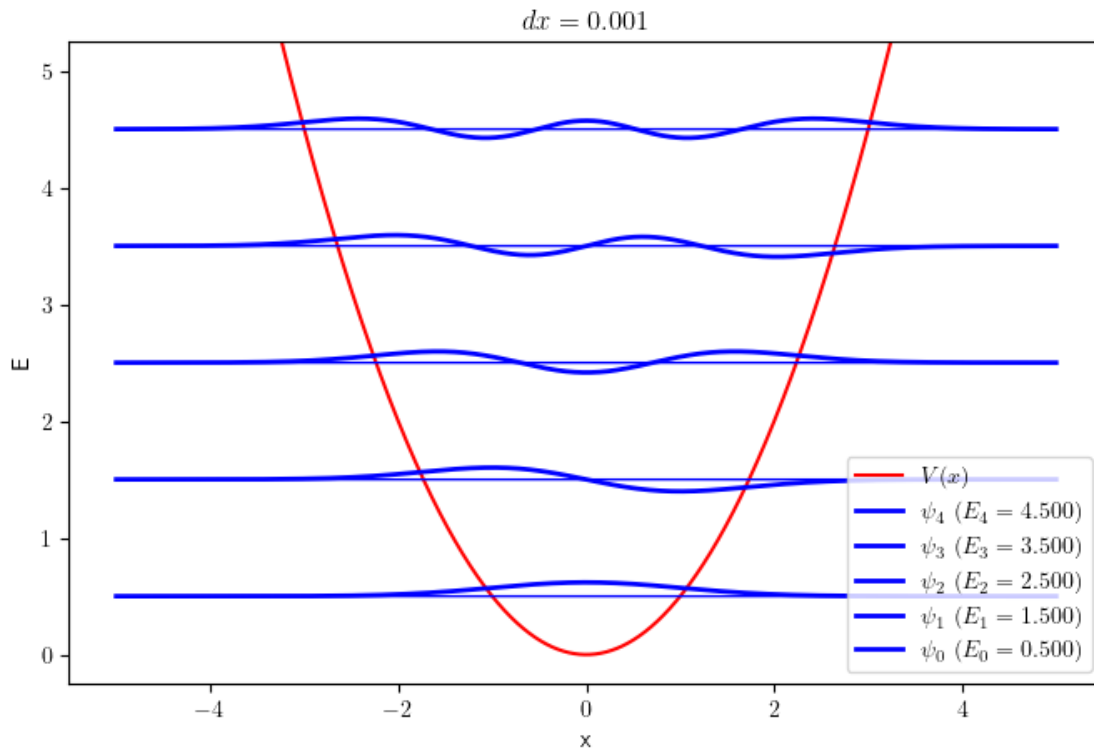
```
[14]: ## Plot an example of the bound states
def plot_bound_states(V, xs, dx, title=''):
    plt.figure()

    pot = V(xs)
    plt.plot(xs, pot, 'r', label='$V(x)$')

    bound_states = get_bound_states(V, xs, dx, n=5)
    for n, (E_comp, psi) in reversed(list(enumerate(zip(*bound_states)))):
        # Plot the exact states
        plt.plot(xs, xs * 0 + E(n), 'b', linewidth=1)
        # Magnify psi for plot purposes
        plt.plot(xs, 5 * psi + E_comp, 'b', label=f'$\\psi_{{{n}}}$')
        plt.plot(xs, (E_comp - E(n)) * 0.03, 'r', label=f'$E_{{{n}}}$', linewidth=2)
```

```
plt.title(title)
plt.xlabel('x')
plt.ylabel('E')
plt.ylim(-0.25, 5.25)
plt.legend(loc='lower right')
plt.gcf().set_size_inches(8, 5)
plt.show()
```

```
[15]: plot_bound_states(V, xs, dx, title=f'${dx=}$')
```

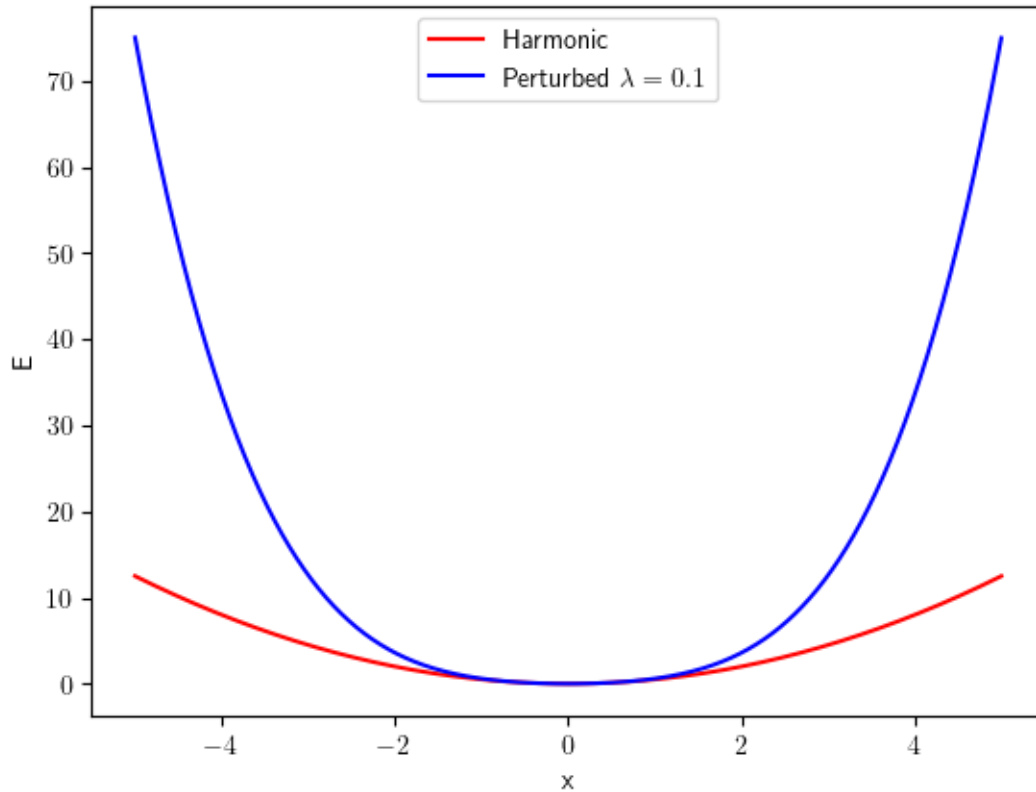


Adding the anharmonicity Now we add a term λx^4 to the potential

```
[16]: def V_ah(l):
        return lambda x: V(x) + perturb(x,l)
def perturb(x,l):
    return l*(x**4)
```

```
[17]: ## Plot and compare the potential
l = 0.1
plt.figure()
plt.plot(xs, V(xs), 'r', label="Harmonic")
```

```
plt.plot(xs, V_ah(1)(xs), 'b', label=f"Perturbed  $\lambda = {1}$ ")
plt.ylabel('E')
plt.xlabel('x')
plt.legend()
plt.show()
```



Now we calculate the first order perturbation to ψ_0 and ψ_1 , namely $\langle \psi_{0,1} | \lambda x^4 | \psi_{0,1} \rangle$.

The exact value for the perturbation theory gap can be calculated analytically using the Hermite polynomial definition or by writing the Hamiltonian in second quantization (creator-annihilator formalism), however here we calculate it numerically in two lines of code using `numpy`:

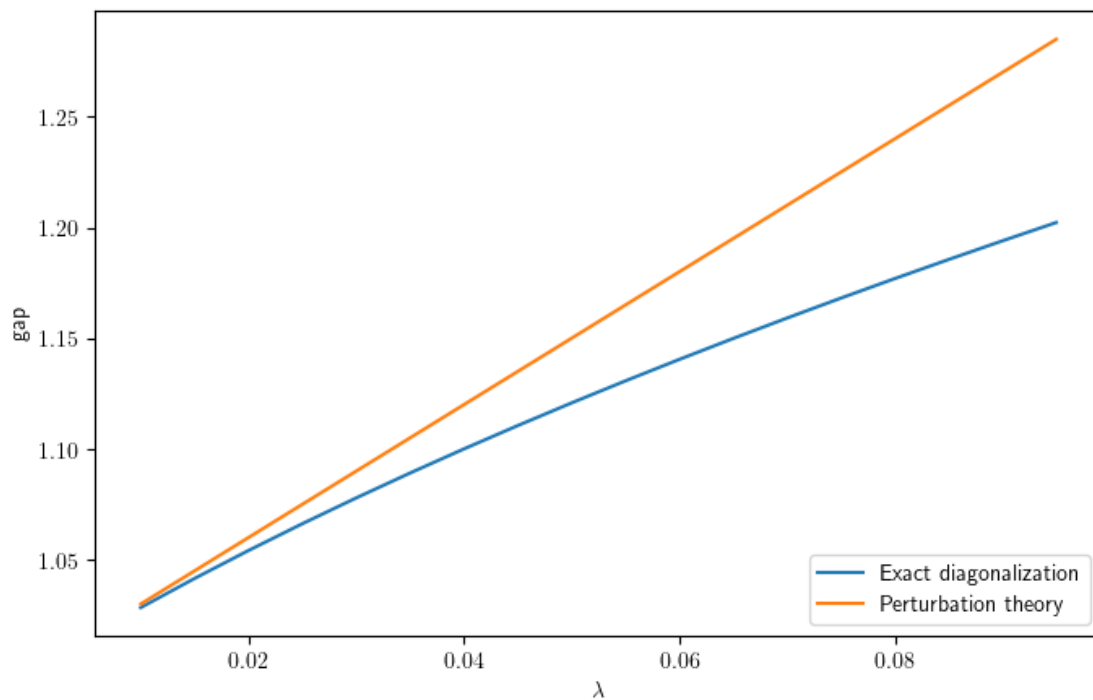
```
[18]: # Now calculate the perturbation
def E0_1(xs,dx,l):
    return psi(xs,0).conj()@(perturb(xs,l)*psi(xs,0)) * dx #<- needed, due to
    ↪ wfn discretization
def E1_1(xs,dx,l):
    return psi(xs,1).conj()@(perturb(xs,l)*psi(xs,1)) * dx

[19]: # Define a vector of lambda values
l_s = np.arange(0.01,0.1,0.005)
```

```
[20]: ## Now for each value of lambda evaluate the gap and compare to the
      ↪ perturbation theory
```

```
gap_comp = []
gap_pert = []
for l in l_s:
    V_ = V_ah(l)
    E_comp = get_bound_eigvals(V_, xs, dx, n=2)
    gap_comp.append(E_comp[1]-E_comp[0])
    gap_pert.append(E(1)+E1_1(xs,dx,l)-E(0)-E0_1(xs,dx,l))
```

```
[21]: plt.figure()
      plt.plot(l_s,gap_comp,label="Exact diagonalization")
      plt.plot(l_s,gap_pert,label="Perturbation theory")
      plt.xlabel('$\\lambda$')
      plt.ylabel('gap')
      plt.legend(loc='lower right')
      plt.gcf().set_size_inches(8, 5)
      plt.show()
```



We can see that, as long as λ is small, perturbation theory agrees with exact diagonalisation, but as the perturbation becomes comparable with the original potential, the accuracy of this approach decreases.

0.1.2 Problem 2.2 - Particles against the wall

In this problem we study the time-evolution of the gaussian wavepacket under the presence of different potentials.

```
[22]: ## Import useful libraries
import numpy as np
import scipy
import scipy.sparse.linalg
from scipy.linalg import eig_tridiagonal
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

w, h = 1920, 1080
dpi = 240
plt.rcParams['figure.figsize'] = w/dpi, h/dpi

from IPython.display import HTML
from tqdm.auto import tqdm
```

We give a function to create an animation given a method to compute the evolution of the wave function

```
[23]: def record_video(wave_step, xs, dx, ts, dt, V=None, title=''):
    '''
    Args:
        wave_step (Callable, args (dt,t)): a function (or class with __call__
        ↪ attribute) to compute the evolution step of the wavefunction
        xs (ndarray): the array of points in which we have
        ↪ discretized space
        dx (float): the discretization step
        ts (ndarray): the array of times for which we
        ↪ calculate time evolution
        dt (float): the time step
        V (function): a function describing the potential
        ↪ (optional)

    '''
    fig, ax1 = plt.subplots()
    if V is not None:
        ax2 = ax1.twinx()
        ax2.plot(xs, V(xs), '--', color='gray')
        for t1 in ax2.get_yticklabels():
            t1.set_color('gray')
        ax2.set_ylim(None, 90) # kind of arbitrary but allows to visualize the
        ↪ angles in last exercise
        ax2.set_ylabel("V", color="gray")
```

```

line_real, = ax1.plot([], [], 'r-', label=r"$\text{Re } \Psi$")
line_imag, = ax1.plot([], [], 'g-', label=r"$\text{Im } \Psi$")
line_abs_sqr, = ax1.plot([], [], 'b-', lw=2, label=r"$|\Psi|^2$")
ax1.set_xlim(min(xs), max(xs))
ax1.set_ylim(-1, 1)
ax1.set_xlabel("x")
ax1.axvline(0, ls="--", color="gray")
ax1.legend(loc='upper right')
ax1.set_title(title)

def update(t):
    psi = wave_step(dt, t)
    line_real.set_data(xs, psi.real)
    line_imag.set_data(xs, psi.imag)
    line_abs_sqr.set_data(xs, np.abs(psi)**2)
    return [line_real, line_imag, line_abs_sqr]

anim = FuncAnimation(fig, update, tqdm(ts), blit=True, interval=1000*dt)
plt.close(fig)
return anim

```

1. “No” Wall We start with the free time evolution (i.e in absence of any potential) of a gaussian wave packet. This is solvable exactly and the exact solution is given by the following function

```

[24]: x_0 = -10.
      k = 5.
      a = 1

def wavepacket(x, t=0, a=a, k=k, x_0=x_0):
    return (
        (np.pi * a**2 / 2)**(-1. / 4) *
        np.sqrt(a**2 / (a**2 + 2j * t)) *
        np.exp(1j*(k*x - k**2 * t / 2)) *
        np.exp(-(x - x_0 - k * t)**2 / (a**2 + 2j * t))
    )

```

Next we define grids for discretizing both the space and time

```

[25]: x_min = -50
      x_max = 50
      x_steps = 5000 # use 5_000 for nice results
      # x_steps = 500 # use 500 for debugging ;- )
      xs, dx = np.linspace(x_min, x_max, x_steps, retstep=True)

      t_min = 0
      t_max = 15
      t_steps = 1000 # use 1_000 for nice results

```

```
# t_steps = 100 # use 100 for debugging ;-)
ts, dt = np.linspace(t_min, t_max, t_steps, retstep=True)
```

We define a dummy potential which is zero everywhere which will be needed for testing our methods without re-defining

```
[26]: # vectorised zero potential
V_free = lambda x: np.zeros_like(x)
```

We can generate an animation for the exact analytical solution for the free particle

```
[27]: def exact_evolution(dt, t):
        return wavepacket(xs, t)
anim = record_video(exact_evolution, xs=xs, dx=dx, ts=ts, dt=dt, title='Exact_
↳Evolution')
anim.save("../solutions_ex02_videos/step1-exact.mp4", fps=1 / dt, dpi=dpi)
#HTML(anim.to_html5_video())
```

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

For discretizing the Hamiltonian we use the same scheme as in the exercise above. For completeness the definition is repeated here. For the later parts of the exercise, use `return_sparse=True` to get a sparse matrix

```
[28]: def hamiltonian(V, xs, dx, return_sparse_matrix=False):
        """compute the main and off-diagonal of the discretized hamiltonian for a_
        ↳given potential V"""
        diag = V(xs) + 1. / dx**2
        offdiag = -0.5 / dx**2 * np.ones(len(xs)-1)
        if return_sparse_matrix:
            return scipy.sparse.diags([diag, offdiag, offdiag], [0, -1, 1])
        else:
            return diag, offdiag
```

Spectral Evolution We start with the first method for simulating the dynamics, the spectral method.

For this we diagonalize hamiltonian finding the set of eigenstates $|\phi_n\rangle$ and eigenvalues E_n such that

$$\hat{H}|\phi_n\rangle = E_n|\phi_n\rangle$$

Given an initial state $|\psi(t_0)\rangle$ in the σ_z basis we need to transform it in to the eigenbasis of the hamiltonian. For this we compute it's coefficients in the eigenbasis given by the overlap $c_n = \langle\phi_n|\psi(t_0)\rangle$ to obtain the representation $|\psi(t_0)\rangle = \sum_n c_n|\phi_n\rangle$

Then the time evolution is given by

$$|\psi(t)\rangle = \sum_n c_n e^{-iE_n(t-t_0)}|\phi_n\rangle$$

which we can compute exactly.

Note that in order to use the spectral method for a free particle we have to make sure that the interval $[x_{min}, x_{max}]$ large enough such that the overlap with the gaussian wavepacket describing it is zero outside.

```
[29]: class SpectralEvolution:
    def __init__(self, H, initial_psi):
        main_diag, offdiag = H
        # use a tridiagonal eigenvalue solver to find the eigenvalues and
        ↪ eigenstates
        self.energies, self.eigenstates = eigh_tridiagonal(main_diag, offdiag)
        # transform initial_psi into the eigenbasis,
        # where the columns of self.eigenstates represent the eigenstates
        # e.g. self.eigenstates[:, 0] is the ground state
        self.initial_psi = np.dot(self.eigenstates.T.conj(), initial_psi)

    def __call__(self, dt, t):
        # define how the self.initial_psi state is updated at time t and return
        ↪ the evolved state
        psi = np.exp(-1j * self.energies * t) * self.initial_psi
        return np.dot(self.eigenstates, psi)
```

We construct the hamiltonian for the free particle

```
[30]: H_free = hamiltonian(V_free, xs=xs, dx=dx, return_sparse_matrix=False)
```

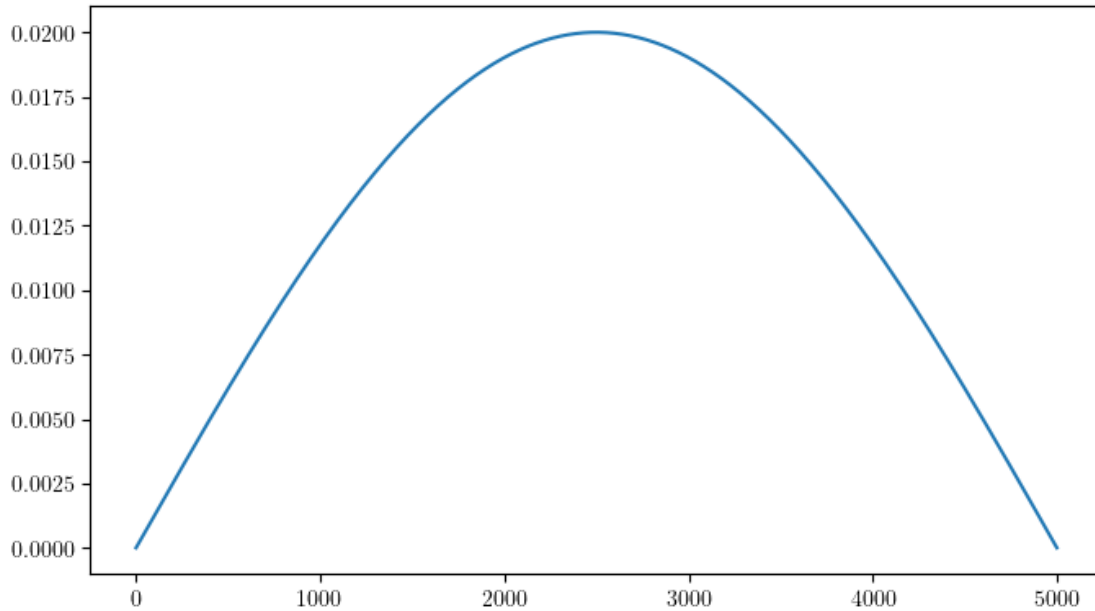
benchmark the spectral evolution:

```
[31]: evolution = SpectralEvolution(H_free, wavepacket(xs))
    %timeit evolution(dt, 0.)
```

76.4 ms ± 41.1 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
[32]: plt.plot(evolution.eigenstates[:, 0])
```

```
[32]: [<matplotlib.lines.Line2D at 0x7f49641728a0>]
```



and run the it to generate an animation

```
[33]: evolution = SpectralEvolution(H_free, wavepacket(xs))
anim = record_video(evolution, xs=xs, dx=dx, ts=ts, dt=dt, title='Spectral')
anim.save("../solutions_ex02_videos/step1-spectral.mp4", fps=1 / dt, dpi=dpi)
HTML(anim.to_html5_video())
```

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

```
[33]: <IPython.core.display.HTML object>
```

Forward Euler For completeness we also include the Forward Euler scheme which is unstable.

```
[34]: import functools

class EulerEvolution:
    def __init__(self, H, initial_psi):
        self._H = H
        self.current_psi = initial_psi.reshape(-1, 1)

        # Note that the cached function will not update if
        # 'self' changes -- if we try to update self._H this
        # approach will not work
        @functools.lru_cache(maxsize=1)
        def _operator(self, dt):
            return - 1j * dt * self._H + scipy.sparse.identity(x_steps)
```

```

def __call__(self, dt, t):
    A = self._operator(dt)
    psi = A.dot(self.current_psi)
    # renormalize state
    self.current_psi = psi / np.linalg.norm(psi) * np.linalg.norm(self.
↪current_psi)
    return self.current_psi

```

```
[35]: H_free_sparse = hamiltonian(V_free, xs=x, dx=dx, return_sparse_matrix=True)
```

```
[36]: evolution = EulerEvolution(H_free_sparse, wavepacket(x))
      %timeit evolution(dt, 0.)
```

36.3 μ s \pm 2.93 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```
[37]: evolution = EulerEvolution(H_free_sparse, wavepacket(x))
      anim = record_video(evolution, xs=x, dx=dx, ts=ts, dt=dt, title='Forward_
↪Euler')
      anim.save("../solutions_ex02_videos/step1-euler.mp4", fps=1 / dt, dpi=dpi)
      #HTML(anim.to_html5_video())
```

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

Forward/Backward unitary scheme The evolution is given by

$$|\psi(t + dt)\rangle = \underbrace{\left(1 + \frac{i dt}{2} \hat{H}\right)^{-1}}_{\text{backward step}} \underbrace{\left(1 - \frac{i dt}{2} \hat{H}\right)}_{\text{forward step}} |\psi(t)\rangle$$

While the forward step only requires the multiplication with a tridiagonal operator, for the backward step we need to solve a system of equations. Since the left-hand side is tridiagonal we can use an efficient solver such as the Thomas algorithm which is what is used by `scipy.linalg.solve_banded` internally.

```
[38]: class StableEvolution:
      def __init__(self, H, initial_psi):
          self._H = H
          self.current_psi = initial_psi

      @functools.lru_cache(maxsize=1)
      def _operator(self, dt):
          # by adding the (lazy) diagonal to H (and not the other way round)
          # we preserve the DIAgonal format of the matrix
          return (- 0.5j * dt * self._H) + scipy.sparse.identity(len(self.
↪current_psi))

      @functools.lru_cache(maxsize=1)
      def _operator_conj(self, dt):
```

```

        return self._operator(dt).conj()

    def __call__(self, dt, t):

        # 1. forward step
        tmp = self._operator(dt).dot(self.current_psi)

        # 2. backward step

        # the operator (I - 0.5i dt H) is not hermitian, so
        # we cannot use a hermitian solver (scipy.linalg.solveh_banded)
        # but have to resort to the normal banded solver (scipy.linalg.
        ↪ solve_banded)
        op = self._operator_conj(dt)
        # extract the main and off diagonal of the tridiagonal
        # operator and store them in the format wanted by the solver
        main_diag = op.diagonal(0)
        ab = np.zeros((3, len(main_diag)), dtype=main_diag.dtype)
        ab[0, 1:] = op.diagonal(1)
        ab[1] = main_diag
        ab[2, :-1] = op.diagonal(-1)
        self.current_psi = scipy.linalg.solve_banded((1,1), ab, tmp)

        return self.current_psi

```

We see that thanks to its sparse implementation the unitary scheme is much faster than the spectral method

```

[39]: evolution = StableEvolution(H_free_sparse, wavepacket(xs))
      %timeit evolution(dt, 0.)

```

155 µs ± 160 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

```

[40]: evolution = StableEvolution(H_free_sparse, wavepacket(xs))
      anim = record_video(evolution, xs=xs, dx=dx, ts=ts, dt=dt, title='Implicit')
      anim.save("../solutions_ex02_videos/step1-implicit.mp4", fps=1 / dt, dpi=dpi)
      HTML(anim.to_html5_video())

```

```

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

```

```

[40]: <IPython.core.display.HTML object>

```

Split-Operator Method In the following we also include an implementation of the split-operator method presented in the appendix of the lecture notes. It is using fast fourier transforms to map from the position space to the momentum space and back.

```

[41]: class SplitOperatorMethod:
      def __init__(self, V, initial_psi, xs):
          self.V = V(xs)

```

```

        self.current_psi = initial_psi
        # compute  $|k|^2$ 
        freqs = np.fft.fftfreq(len(xs))
        self.neg_laplace = (2 * np.pi * (len(xs) - 1) / (max(xs) - min(xs)) *
↪freqs)**2

    @functools.lru_cache(maxsize=1)
    def _exp_V(self, dt):
        #  $\exp(-i \, dt/2 \, V(x))$ 
        return np.exp(-0.5j * dt * self.V)

    @functools.lru_cache(maxsize=1)
    def _exp_neg_laplace(self, dt):
        #  $\exp(-i \, dt \, |k|^2/2)$ 
        return np.exp(-1j * dt * self.neg_laplace / 2)

    def __call__(self, dt, t):
        exp_V = self._exp_V(dt)
        psi = exp_V * self.current_psi
        psi = np.fft.fft(psi) # fourier transform
        psi = self._exp_neg_laplace(dt) * psi
        psi = np.fft.ifft(psi) # inverse fourier transform
        self.current_psi = exp_V * psi
        return self.current_psi

```

```

[42]: evolution = SplitOperatorMethod(V_free, wavepacket(xs), xs)
      %timeit evolution(dt, None)

```

62 μ s \pm 121 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```

[43]: evolution = SplitOperatorMethod(V_free, wavepacket(xs), xs)
      anim = record_video(evolution, xs=xs, dx=dx, ts=ts, dt=dt,
↪title='Split-Operator Method')
      anim.save("../solutions_ex02_videos/step1-split.mp4", fps=1 / dt, dpi=dpi)
      #HTML(anim.to_html5_video())

```

```

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

```

2. Infinite Wall Next we can study how the wavepacket is reflected off a infinitely high wall.

The tridiagonal Hamiltonian for the kinetic part that we set up above automatically implements vanishing boundary conditions at x_{\min} and x_{\max} .

Thus we merely set $x_{\max} = 0$:

```

[44]: x_min = -45
      x_max = 0
      x_steps = 1000
      xs, dx = np.linspace(x_min, x_max, x_steps, retstep=True)

```



```
V_free = lambda x: np.zeros_like(x)
H_free = hamiltonian(V_free, xs=xs, dx=dx, return_sparse_matrix=False)

anim = record_video(SpectralEvolution(H_free, wavepacket(xs)), xs=xs, dx=dx,
    ↪ts=ts, dt=dt)
anim.save("../solutions_ex02_videos/step2-spectral.mp4", fps=1 / dt, dpi=dpi)
#HTML(anim.to_html5_video())
```

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

3. Tilted Wall For the last part we study what happens when the particle hits the tilted wall

$$V(x) = \begin{cases} 0 & x < 0 \\ \tan(\theta) x & x \geq 0 \end{cases}$$

```
[45]: x_min = -20
x_max = 50
x_steps = 1000
xs, dx = np.linspace(x_min, x_max, x_steps, retstep=True)
```

```
[46]: def V_tilted(theta_deg):
    m = np.tan(theta_deg*2*np.pi/360)
    return lambda x: np.maximum(0, m*x)

def H_tilted(theta_deg, xs, dx):
    return hamiltonian(V_tilted(theta_deg), xs=xs, dx=dx,
    ↪return_sparse_matrix=True)
```

In particular we look at how the velocity

$$v(t) = \int_{-\infty}^{\infty} \text{Im}[\Psi(x, t)^* \frac{d}{dx} \Psi(x, t)] dx$$

of the wave packet evolves over time for different tilting angles θ

```
[47]: def velocity(psi, dx):
    # compute the gradient of psi with finite differences
    grad_psi = np.diff(psi)/dx
    # integrate using the trapezoidal rule
    psi_mean = (psi[1:] + psi[:-1])/2
    return (psi_mean.conj() * grad_psi).imag.sum() * dx
```

```
[48]: def velocities(theta_deg):
    evol = StableEvolution(H_tilted(theta_deg, xs=xs, dx=dx), wavepacket(xs))
    vs = [velocity(wavepacket(xs, 0), dx=dx)]
    for t in ts[1:]:
```

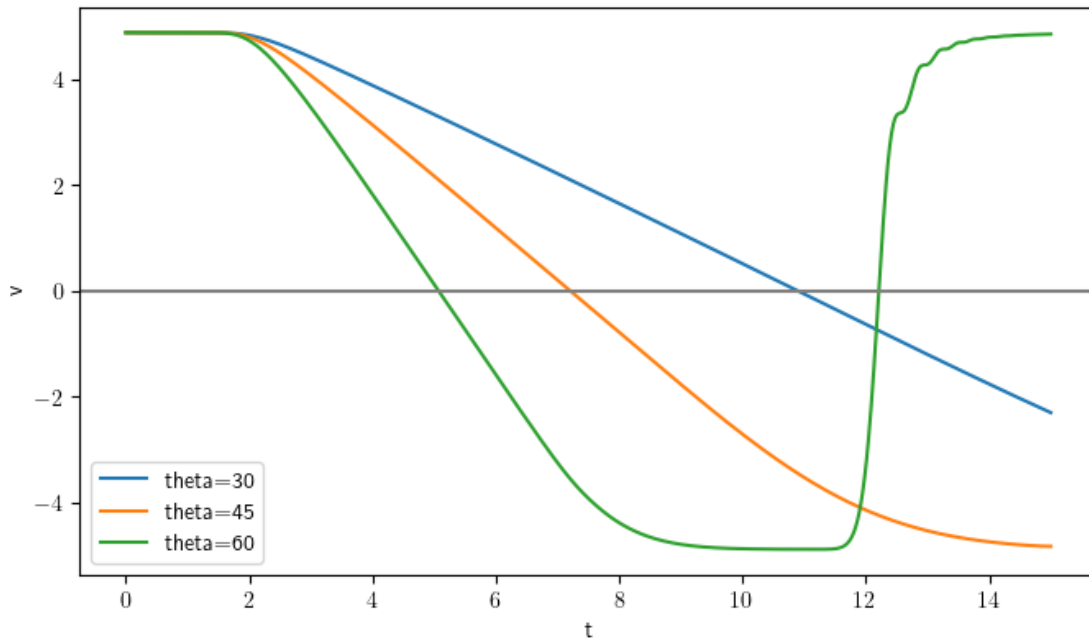
```

        psi = evol(dt, t)
        vs.append(velocity(psi, dx=dx))
    return vs

angles = [30, 45, 60]

plt.figure()
plt.xlabel("t")
plt.ylabel("v")
for theta_deg in angles:
    vs = velocities(theta_deg)
    plt.plot(ts, vs, label=f"theta={theta_deg}" )
plt.legend()
plt.axhline(0, color="gray")
plt.show()
#plt.savefig("step3-velocities.png", dpi=300)

```



```

[49]: animations = []
for theta_deg in angles:
    evolution = StableEvolution(H_tilted(theta_deg, xs=xs, dx=dx),
    ↪ wavepacket(xs))
    anim = record_video(evolution, V = V_tilted(theta_deg), xs=xs, dx=dx,
    ↪ ts=ts, dt=dt)
    anim.save(f'../solutions_ex02_videos/step3-implicit-{theta_deg}.mp4', fps=1,
    ↪ dt, dpi=dpi)

```

```
animations.append(anim)
```

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

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

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

```
[50]: #HTML(animations[0].to_html5_video())
```

```
[51]: #HTML(animations[1].to_html5_video())
```

```
[52]: #HTML(animations[2].to_html5_video())
```