HW2

December 5, 2022

1 Quantum Dynamics

1.1 Time-Dependent Schrödinger Equation

We start off with solving the time-dependent schrödinger equation by discretizing the dependent variable x, so we instead obtain a matrix equation as following

$$H^{N\times N}\Psi^{N\times 1} = i\hbar\frac{d}{dt}\Psi^{N\times 1}$$

where the superscripts are the matrix dimension.

Then the solution to this equation can be written as

$$\vec{\Psi}(t+\Delta t) = \mathsf{U}(\Delta t)\,\vec{\Psi}(t) \tag{1}$$

Now if the Hamiltonian matrix H is time-independent, the exact expression for the time-evolution operator is

$$\mathsf{U}(\Delta t) = e^{-i\mathsf{H}\Delta t/\hbar} \tag{2}$$

But for small Δt, the time-evolution operator can be approximated with Taylor expansion of e as

$$\mathsf{U}(\Delta t) \approx 1 - i\mathsf{H}\,\frac{\Delta t}{\hbar}$$

or more preferably with Cayley's form,

$$\mathsf{U}(\Delta t) \approx \frac{1 - \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}}{1 + \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}} \tag{3}$$

And substituting this into the solution of the the TDSE, we get

$$\left(1 + \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}\right)\vec{\Psi}(t+\Delta t) = \left(1 - \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}\right)\vec{\Psi}(t) \tag{4}$$

Now we need to check that the Eq. (3) is accurate to the second order. We expand Eqs. (1) & (3) as power series in Δt, for the first one we have

$$\mathsf{U}(\Delta t) = 1 - i\mathsf{H}\,\frac{\Delta t}{\hbar} + \frac{1}{2}\left(i\mathsf{H}\,\frac{\Delta t}{\hbar}\right)^2 - ...$$

and for the (3)

$$\mathsf{U}(\Delta t) \approx \frac{1 - \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}}{1 + \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}} = \left(1 - \frac{i}{2} \frac{\Delta t \mathsf{H}}{\hbar}\right) \left(1 - \frac{i}{2} \frac{\Delta t \mathsf{H}}{\hbar} - \frac{1}{2} \left(\frac{\Delta t \mathsf{H}}{\hbar}\right)^2 + \dots\right) = 1 - i \frac{\Delta t \mathsf{H}}{\hbar} - \frac{1}{2} \left(\frac{\Delta t \mathsf{H}}{\hbar}\right)^2 + \dots$$

which agrees up to the Δt^2

Now to check that U is unitary matrix, by definition, we require that $\mathsf{U}\mathsf{U}^\dagger = 1$

$$\mathsf{U}\mathsf{U}^\dagger = \left(\frac{1 - \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}}{1 + \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}}\right) \left(\frac{1 + \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}}{1 - \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}}\right) = 1$$

1.2 Construct The Hamiltonian Matrix

We construct the the Hamiltonian matrix H for $N + 1 = 100$ spatial grid points, and the spatial boundaries with dimensionless length of $\xi = \pm 10$. With setting, $m = 1$, $\omega = 1$, and $\hbar = 1$.

1.3 Lowest Two Eigenvalues of H

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from scipy.linalg import eigh_tridiagonal, solve
from scipy.special import hermite
from scipy.sparse import diags

from matplotlib_inline.backend_inline import set_matplotlib_formats

set_matplotlib_formats('pdf', 'svg')
plt.style.use('seaborn')
plt.rcParams |= {
    'text.usetex': True,
    'figure.figsize': (14, 5)
}
```

[ ]:
```python
m = omega = hbar = 1
Nx = 100
xmin = -10
xmax = 10
dx = (xmax - xmin) / (Nx + 1)

# The second derivative operator
D2 = (
    np.diag([-2] * Nx)
    + np.diag(np.ones(Nx - 1), 1)
    + np.diag(np.ones(Nx - 1), -1)
) / dx**2
```

```python
# The position operator
X = np.diag(np.linspace(xmin, xmax, Nx))

# The Hamiltonian
H = -1 / (2 * m) * D2 + 1 / 2 * m * omega * X**2

# diagonal elements of H
d = H.diagonal(0)
# above-diagonal elements of H
e = H.diagonal(1)
# Find the eigenvalues and eigenvectors of H
# This algorithm is much faster than using H entirly
eigenvalues, eigenvectors = eigh_tridiagonal(
    d, e, select="i", select_range=(0, 1)
)

# check if the first eigenvector is normalized, which is by default
np.round(np.dot(eigenvectors[:, 0], eigenvectors[:, 0]), 10)
```

[ ]: 1.0

The exact eigenvalues for the harmonic oscillator are,

$$E_n = \left(n + \frac{1}{2}\right) \hbar\omega$$

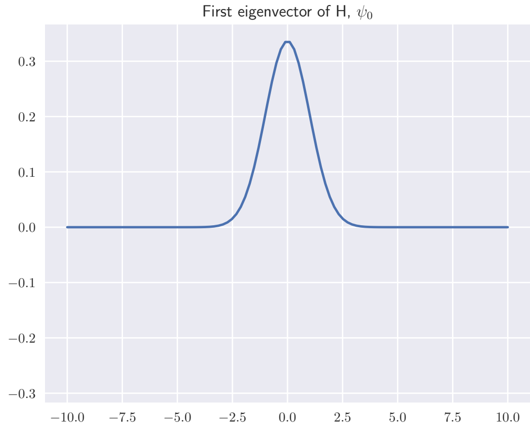
In the above units, we have $E_0 = 0.5$ and $E_1 = 1.5$

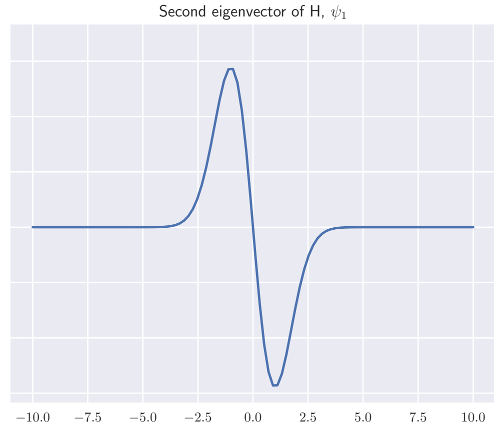
```python
print(f'The first eigenvalue is {eigenvalues[0]:.3f} with error␣
 ↪{(eigenvalues[0] - 0.5) / 0.5 *100 :.2f}%')
print(f'The second eigenvalue is {eigenvalues[1]:.3f} with error␣
 ↪{(eigenvalues[1] - 1.5) / 1.5 *100 :.2f}%')
```

```
The first eigenvalue is 0.509 with error 1.76%
The second eigenvalue is 1.524 with error 1.59%
```

1.4 Plot The Eigenvector

```python
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
x = X.diagonal(0)

ax1.plot(x, eigenvectors[:, 0])
ax2.plot(x, eigenvectors[:, 1])
ax1.set_title('First eigenvector of $\mathsf{H}$, $\psi_0$')
ax2.set_title('Second eigenvector of $\mathsf{H}$, $\psi_1$')

plt.show()
```





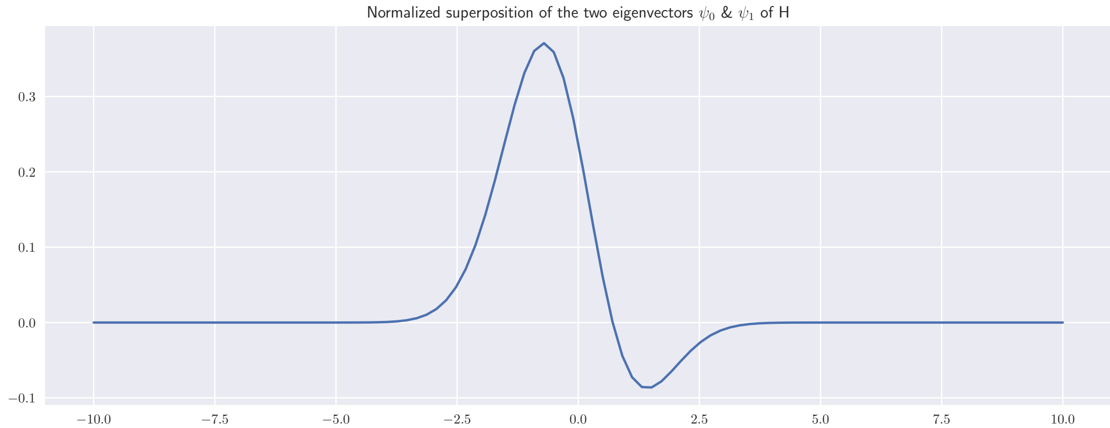
1.5 Evolve The Wave Function

Now we need to show an animation of the evolving wave function using (4) from time $t = 0$ to $t = 4\pi/\omega$. We will take $\Psi(0)$ to be

$$\Psi(0) = \frac{\psi_0 + \psi_1}{\sqrt{2}}$$

Note for the exp approximation to hold, the time steps should be at least $N_2 = 100$.

[ ]:
```
# Taking this if time equals 0
Psi0 = (eigenvectors[:, 0] + eigenvectors[:, 1]) / np.sqrt(2)

plt.plot(x, Psi0)
plt.title('Normalized superposition of the two eigenvectors $\psi_0$ \&␣
  ↪$\psi_1$ of $\mathsf{H}$')
plt.show()
```



1.5.1 Construct U

From (3) we build the following

```
[ ]: Nt = 1000
     dt = 2 * np.pi / omega / Nt

     # The denominator in Cayley's form
     U_plus = np.identity(Nx) + 0.5j * H * dt
     # The numerator in Cayley's form
     U_minus = np.identity(Nx) - 0.5j * H * dt
```

1.5.2 Solve TDSE

Now we solve numerically the time-dependent Schrödinger equation, that is solving the matrix equation in (4):

$$\left(1 + \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}\right)\vec{\Psi}(t + \Delta t) = \left(1 - \frac{1}{2}i\frac{\Delta t}{\hbar}\mathsf{H}\right)\vec{\Psi}(t) \tag{4}$$

We imagine it as in the form of

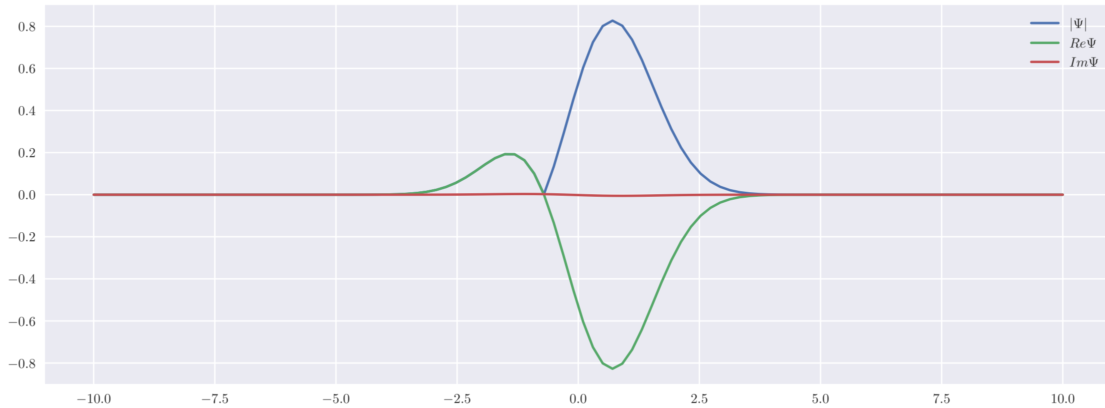
$$\mathsf{M}\vec{x} = \vec{b}$$

```
[ ]: fig, ax = plt.subplots()

     (graph1,) = ax.plot(x, np.abs(Psi0), label=r"$|\Psi|$")
     (graph2,) = ax.plot(x, np.real(Psi0), label=r"$Re{\Psi}$")
     (graph3,) = ax.plot(x, np.imag(Psi0), label=r"$Im{\Psi}$")
     ax.set_ylim([-0.3, 0.4])
     ax.legend()

     # a list of soltion of psi(t)
     psi_solutions = [Psi0]


     def update(i):
         # solve the linear matrix eq
         b = U_minus @ psi_solutions[i]
         psi_solutions.append(solve(U_plus, b))

         graph1.set_ydata(np.abs(psi_solutions[-1]))
         graph2.set_ydata(np.real(psi_solutions[-1]))
         graph3.set_ydata(np.imag(psi_solutions[-1]))

         return graph1, graph2, graph3


     animation = FuncAnimation(
         fig,
         update,
```

```
    frames=Nt,
    blit=True,
)
animation.save("evolve_approx.mp4", fps=100, dpi=150)
```

1.5.3 Evolve the Exact Solution

[ ]:
```
def psi(n, x):
    # The harmonic osillator wavefunction
    return (
        (m * omega / np.pi / hbar) ** 0.25
        * (1 / np.sqrt(2**n * np.prod(np.arange(1, n + 1))))
        * hermite(n)(x)
        * np.exp(-0.5 * ((np.sqrt(m * omega / hbar) * x) ** 2))
    )

def Psi(i):
    psi0 = psi(0, x)
    psi1 = psi(1, x)
    # the exact superposition
    return (
        psi0 * np.exp(-0.5j * omega * i * dt)
        + psi1 * np.exp(-1.5j * omega * i * dt)
    ) / np.sqrt(2)

fig, ax = plt.subplots()
(graph1,) = ax.plot(x, np.abs(Psi(0)), label=r"$|\Psi|$")
(graph2,) = ax.plot(x, np.real(Psi(0)), label=r"$Re{\Psi}$")
(graph3,) = ax.plot(x, np.imag(Psi(0)), label=r"$Im{\Psi}$")
ax.set_ylim([-0.9, 0.9])
ax.legend()

def update(i):
    graph1.set_ydata(np.abs(Psi(i)))
    graph2.set_ydata(np.real(Psi(i)))
    graph3.set_ydata(np.imag(Psi(i)))

    return graph1, graph2, graph3

animation = FuncAnimation(
    fig,
    update,
    frames=Nt,
    blit=True,
)
animation.save("evolve_exact.mp4", fps=100, dpi=150)
```



1.6 Time-Dependent Hamiltonian

Now we need to use the above techneuqes to observe the time evolving of time-dependent Schrödinger equation and where the Hamiltonian *does* depend on time. Instead of (4), we will use the following and evaluating H at the midpoint of each time step

$$\left[1 + \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}\left(t + \frac{\Delta t}{2}\right)\right] \vec{\Psi}(t + \Delta t) = \left[1 - \frac{1}{2} i \frac{\Delta t}{\hbar} \mathsf{H}\left(t + \frac{\Delta t}{2}\right)\right] \vec{\Psi}(t)$$

Consider the driving harmonic oscillator

$$f(t) = A \sin(\Omega t)$$

where $A = 1$ is a constant with units of length and Ω is the driving frequency.

[ ]:
```
# The driving frequency value
Omega = omega / 5
A = 1

def f(t):
    # driving harmonic function
    return A * np.sin(Omega * t)

# The time-dep Hamiltonian
def H(t):
    return (
        -0.5 * hbar**2 / m * D2
        + 0.5 * m * omega**2 * X**2
        - m * omega**2 * f(t) * X
    )

# The numerator in Cayley's form
def U_plus(t):
    return np.identity(Nx) + 1j * dt / 2 * H(t + dt / 2)
```

```python
# The denominator in Cayley's form
def U_minus(t):
    return np.identity(Nx) - 1j * dt / 2 * H(t + dt / 2)

# diagonal elements of H(0)
d = H(0).diagonal(0)
# above-diagonal elements of H(0)
e = H(0).diagonal(1)

# Find the eigenvalues and eigenvectors of H at t = 0
eigenvalues, eigenvectors = eigh_tridiagonal(
    d, e, select="i", select_range=(0, 1)
)
```

[ ]:

```python
# The wavefunction of H at t = 0
Psi0 = eigenvectors[:, 0]
t = np.linspace(0, 2 * np.pi / Omega, Nt)

fig, ax = plt.subplots()
(graph1,) = ax.plot(x, np.abs(Psi0), label=r"$|\Psi|$")
(graph2,) = ax.plot(x, np.real(Psi0), label=r"$Re{\Psi}$")
(graph3,) = ax.plot(x, np.imag(Psi0), label=r"$Im{\Psi}$")
(graph4,) = ax.plot(x, np.abs(psi(0, x-f(t[0]))), label=r"$|\psi_0|$")
ax.set_ylim([-0.4, 0.4])
ax.legend()

# a list of soltion of psi(t)
psi_solutions = [Psi0]

def update(i):
    # solve the linear matrix eq
    b = U_minus(t[i]) @ psi_solutions[i]
    psi_solutions.append(solve(U_plus(t[i]), b))

    graph1.set_ydata(np.abs(psi_solutions[-1]))
    graph2.set_ydata(np.real(psi_solutions[-1]))
    graph3.set_ydata(np.imag(psi_solutions[-1]))
    graph4.set_ydata(psi(0, x-f(t[i])))

    return graph1, graph2, graph3, graph4

animation = FuncAnimation(
    fig,
    update,
    frames=len(t),
    blit=True,
```

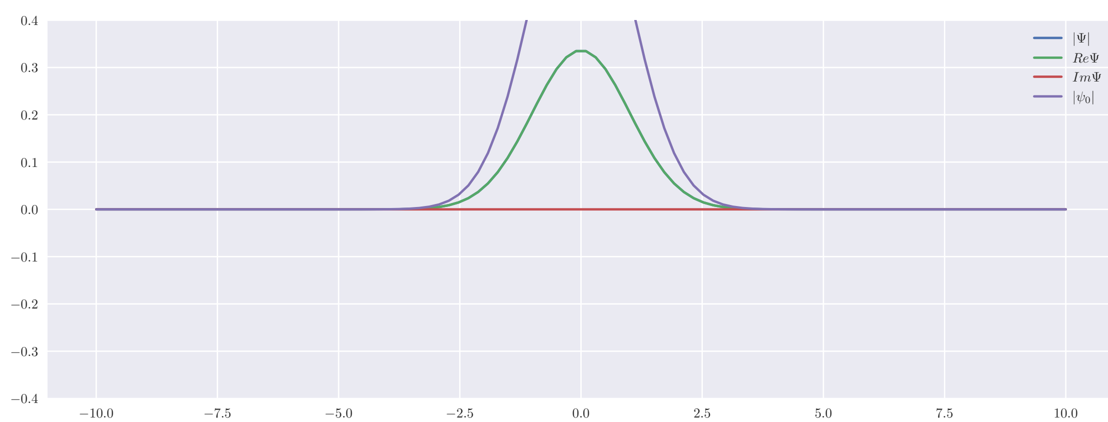
```
)
animation.save("driven_evolve_approx_and_exact.mp4", fps=500, dpi=150)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [30], line 28
     24     # graph4.set_ydata(psi(0, x-f(t[i])))
     26     return graph1, graph2, graph3, graph4
---> 28 animation = FuncAnimation(
     29     fig,
     30     update,
     31     frames=len(t),
     32     blit=True,
     33 )
     34 animation.save("driven_evolve_approx_and_exact.mp4", fps=500, dpi=150)

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:1634, in FuncAnimation.__init__(self, fig, func,␣
 ↪frames, init_func, fargs, save_count, cache_frame_data, **kwargs)
   1631 # Needs to be initialized so the draw functions work without checking
   1632 self._save_seq = []
-> 1634 super().__init__(fig, **kwargs)
   1636 # Need to reset the saved seq, since right now it will contain data
   1637 # for a single frame from init, which is not what we want.
   1638 self._save_seq = []

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:1396, in TimedAnimation.__init__(self, fig, interval,␣
 ↪repeat_delay, repeat, event_source, *args, **kwargs)
   1394 if event_source is None:
   1395     event_source = fig.canvas.new_timer(interval=self._interval)
-> 1396 super().__init__(fig, event_source=event_source, *args, **kwargs)

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:883, in Animation.__init__(self, fig, event_source,␣
 ↪blit)
    880 self._close_id = self._fig.canvas.mpl_connect('close_event',
    881                                               self._stop)
    882 if self._blit:
--> 883     self._setup_blit()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:1197, in Animation._setup_blit(self)
   1194 self._drawn_artists = []
   1195 self._resize_id = self._fig.canvas.mpl_connect('resize_event',
   1196                                                self._on_resize)
-> 1197 self._post_draw(None, self._blit)
```

```
File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/animation.py:1150, in Animation._post_draw(self, framedata, blit)
   1148     self._blit_draw(self._drawn_artists)
   1149 else:
-> 1150     self._fig.canvas.draw_idle()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/backend_bases.py:2060, in FigureCanvasBase.draw_idle(self, *args,␣
↪**kwargs)
   2058 if not self._is_idle_drawing:
   2059     with self._idle_draw_cntx():
-> 2060         self.draw(*args, **kwargs)

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/backends/backend_agg.py:436, in FigureCanvasAgg.draw(self)
    432 # Acquire a lock on the shared font cache.
    433 with RendererAgg.lock, \
    434      (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
    435       else nullcontext()):
--> 436     self.figure.draw(self.renderer)
    437     # A GUI class may be need to update a window using this draw, so
    438     # don't forget to call the superclass.
    439     super().draw()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/artist.py:74, in _finalize_rasterization.<locals>.
↪draw_wrapper(artist, renderer, *args, **kwargs)
     72 @wraps(draw)
     73 def draw_wrapper(artist, renderer, *args, **kwargs):
---> 74     result = draw(artist, renderer, *args, **kwargs)
     75     if renderer._rasterizing:
     76         renderer.stop_rasterizing()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/artist.py:51, in allow_rasterization.<locals>.draw_wrapper(artist,␣
↪renderer)
     48     if artist.get_agg_filter() is not None:
     49         renderer.start_filter()
---> 51     return draw(artist, renderer)
     52 finally:
     53     if artist.get_agg_filter() is not None:

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/figure.py:2855, in Figure.draw(self, renderer)
   2852 finally:
   2853     self.stale = False
-> 2855 self.canvas.draw_event(renderer)
```

```
File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/backend_bases.py:1779, in FigureCanvasBase.draw_event(self,␣
 ↪renderer)
   1777 s = 'draw_event'
   1778 event = DrawEvent(s, self, renderer)
-> 1779 self.callbacks.process(s, event)

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/cbook/__init__.py:292, in CallbackRegistry.process(self, s, *args,␣
 ↪**kwargs)
    290 except Exception as exc:
    291     if self.exception_handler is not None:
--> 292         self.exception_handler(exc)
    293     else:
    294         raise

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/cbook/__init__.py:96, in _exception_printer(exc)
     94 def _exception_printer(exc):
     95     if _get_running_interactive_framework() in ["headless", None]:
---> 96         raise exc
     97     else:
     98         traceback.print_exc()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/cbook/__init__.py:287, in CallbackRegistry.process(self, s, *args,␣
 ↪**kwargs)
    285 if func is not None:
    286     try:
--> 287         func(*args, **kwargs)
    288     # this does not capture KeyboardInterrupt, SystemExit,
    289     # and GeneratorExit
    290     except Exception as exc:

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:907, in Animation._start(self, *args)
    904 self._fig.canvas.mpl_disconnect(self._first_draw_id)
    906 # Now do any initial draw
--> 907 self._init_draw()
    909 # Add our callback for stepping the animation and
    910 # actually start the event_source.
    911 self.event_source.add_callback(self._step)

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
 ↪matplotlib/animation.py:1696, in FuncAnimation._init_draw(self)
   1688         warnings.warn(
   1689             "Can not start iterating the frames for the initial draw. "
   1690             "This can be caused by passing in a 0 length sequence "
```

```
    (…)
   1693             "it may be exhausted due to a previous display or save."
   1694         )
   1695         return
-> 1696     self._draw_frame(frame_data)
   1697 else:
   1698     self._drawn_artists = self._init_func()

File /Volumes/SSD/projects/python/physics/env/lib/python3.10/site-packages/
↪matplotlib/animation.py:1718, in FuncAnimation._draw_frame(self, framedata)
   1714 self._save_seq = self._save_seq[-self.save_count:]
   1716 # Call the func with framedata and args. If blitting is desired,
   1717 # func needs to return a sequence of any artists that were modified.
-> 1718 self._drawn_artists = self._func(framedata, *self._args)
   1720 if self._blit:
   1722     err = RuntimeError('The animation function must return a sequence '
   1723                        'of Artist objects.')

Cell In [30], line 18, in update(i)
     16 def update(i):
     17     # solve the linear matrix eq
---> 18     b = U_minus(t[i]) @ psi_solutions[i]
     19     psi_solutions.append(solve(U_plus(t[i]), b))
     21     graph1.set_ydata(np.abs(psi_solutions[-1]))

Cell In [9], line 27, in U_minus(t)
     26 def U_minus(t):
---> 27     return np.identity(Nx) - 1j * dt / 2 * H(t + dt / 2)

TypeError: 'numpy.ndarray' object is not callable
```



With driving frequency value of $\Omega = \omega/5$, the numerical solution is very close to the instantaneous ground state. Meaning that, this is an adiabatic process.