Harmonic Oscillator

- **Author:** Ariel Quelal
- **Date:** 11/16/2025
- **Time spent on this assignment:** 48h

Our goal in this assignment is to look at and think about the simple Harmonic oscillator.

In [1]:
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import math
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
import numpy.polynomial.hermite as Herm
import math
import scipy.optimize


import matplotlib.animation as animation
from IPython.display import HTML
def resetMe(keepList=[]):
    ll=%who_ls
    keepList=keepList+['FuncAnimation','Herm','HTML','resetMe','scipy','np','plt
    for iiii in keepList:
        if iiii in ll:
            ll.remove(iiii)
    for iiii in ll:
        jjjj="^"+iiii+"$"
        %reset_selective -f {jjjj}
    ll=%who_ls
    plt.rcParams.update({"font.size": 14})
    return
resetMe()
import datetime;datetime.datetime.now()
```

Out[1]:
```
datetime.datetime(2025, 11, 18, 11, 47, 50, 84341)
```

In this assignment, we will consistently use grids and observables that we will set up in this way. This is very similar to how you set things for the particle in the box assignment.

In [2]:
```
def SetupGrid(L,delta_x):
    n=int(round(L/delta_x))+1
    xs=np.linspace(-L/2,L/2,n,endpoint=True)
    return xs


def SetupObservables(xs,delta_x):
    X=np.diag(xs)
    P=(np.diag([-1.j/(2*delta_x) for i in range(len(xs)-1)],k=1)+np.diag([1.j/(2
    P2=np.zeros_like(P)
    for i in range(len(xs)):
        P2[i,i]=2.0/delta_x**2
        if i+1<len(xs):
            P2[i,i+1]=-1.0/delta_x**2
            P2[i+1,i]=-1.0/delta_x**2
    return X,P,P2


def update(frame, skip, xs, positions=None, potential=[], max_value=0, energy=0,
    plt.cla()  # Clear the current plot
    if type(positions)==np.ndarray:
        pass
    plt.plot(xs, np.abs(arrays[::skip][frame])**2)  # Plot the current array
    plt.ylim(0, 1.5*max_value)  # Set the y-axis limit
    plt.xlabel('x')
    plt.ylabel('Value')
    plt.title(f'Frame {skip*frame+1}/{len(arrays)}')  # Display the frame number
```

Exercise 1. The Uncertainty Principle

In this exercise, we'd like to think about the uncertainty principle.

a. The Fourier Basis

The uncertainty principle tells us that you can't simultaneously know a particle's position and momentum. It's typically quantified as

$$\Delta X \Delta P > \frac{1}{2}$$

Let's think for a moment about what it means to be certain or uncertain about a particles position. If you are uncertain about the position of a particle, what this means is that if you measure that particle many times, you get very different answers for the x position of a particle. If you are certain about the position of a particle this means that most measurements come out concentrated in a particular region so even before measuring, we could feel reasonably certain where the particle is in x.

Here is an example of a wave-function which you are reasonably certain about it's position:

$$\Psi = N \exp[-\alpha(x-4)^6]$$

where N is the normalization and $\alpha = 100$.

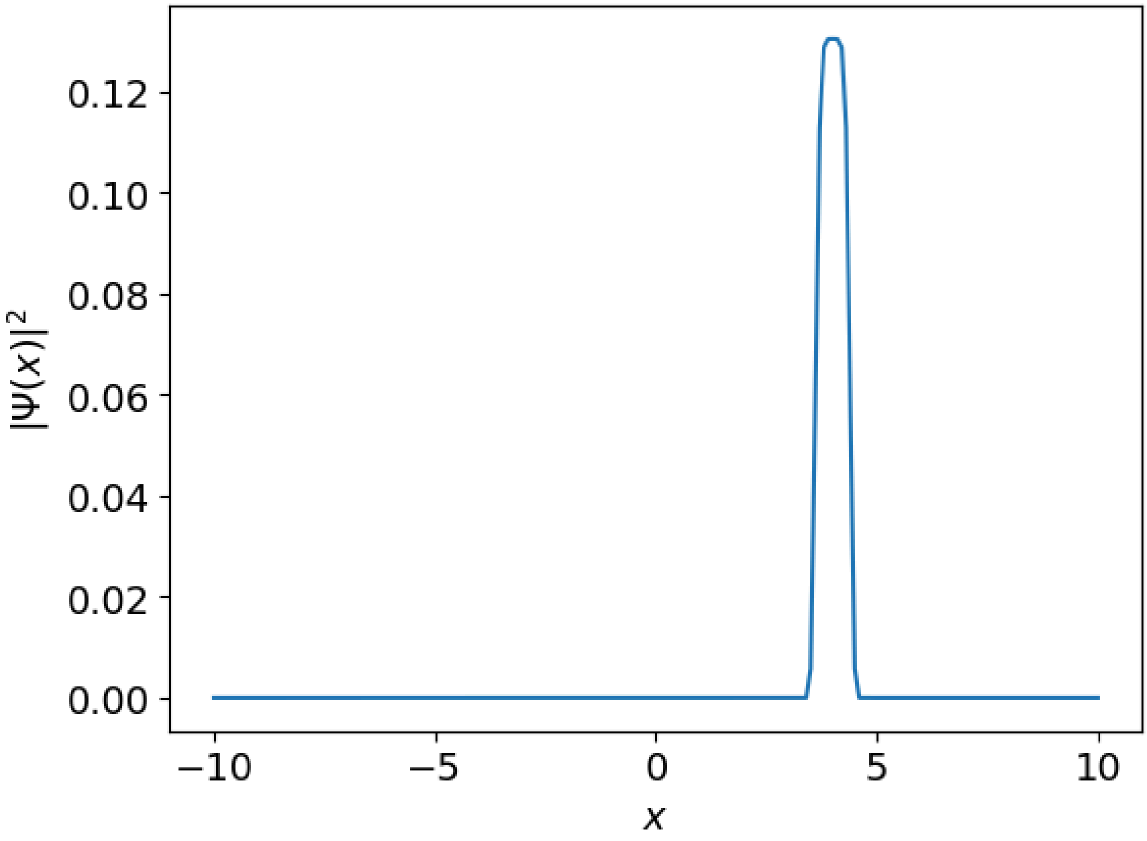
After setting up your grid and observables:

```
    delta_x=0.1
    xs=SetupGrid(20,delta_x)
    X,P,P2=SetupObservables(xs,delta_x)
```

go ahead and plot the probability (absolute value squared of the wave-function) of getting different x values for this wave-function rememembering to label the x-axis as `plt.xlabel("x")` and the y-axis as `plt.ylabel("probability density")`. From this plot, you should be able to see that if you measure x over and over again if will always be getting an answer somewhat around 4.

? Answer (start)

In [3]:
```
delta_x=0.1
L=20
xs=SetupGrid(L,delta_x)
X,P,P2=SetupObservables(xs,delta_x)

#Defining the particle state
def waveFunction(x, alpha=100):
    return np.exp(-alpha*(x-4)**6)

psi=waveFunction(xs)

#Normalizing the wave function and casting it into a column vector

def normalize(psi):
    #psi = np.asarray(psi).reshape(-1, 1)                 #converts python lists
    norm = np.sqrt((psi.conjugate() @ psi.T).item())
    return psi / norm

psi=normalize(psi)

#Plotting the probability density function
def plotPDfunction(psi,xs):
    y=np.abs(psi)**2
    x=xs
    plt.plot(x,y)
    plt.xlabel("$x$")
    plt.ylabel("$|\Psi (x)|^2$")
    #plt.show()

plotPDfunction(psi,xs)
```



✓ Answer (end)

We are now hoping that there's also a pretty narrow range of where the particle can be in momentum P - i.e. if we measure the momentum over and over again would we get essentially the same value. It's hard to look at our plot and figure this out at the moment. That's because the wave-function is currently plotted in the x-basis. We could also plot it in the momentum basis. If we did that, then we could look at the wave-function squared and see how spread out it is in momentum (and so whether we would expect to get the same momentum if we measured it).

So we'd like to replot the wave-function as a function of p and not x. Mind you, this is the same wave-function. It's just a different way of representing it. If I show you a curve and I don't tell label the axis so you know whether you are in the x basis or p basis, you don't really know what you're looking at.

To change a basis you mutiply by a unitary matrix that corresponds to that basis change. We need the unitary matrix that multiplies a wave-function in the x-basis and converts it to the p-basis. Then we can plot it in the p-basis. You can get this unitary (along with the basis of p-points) by doing

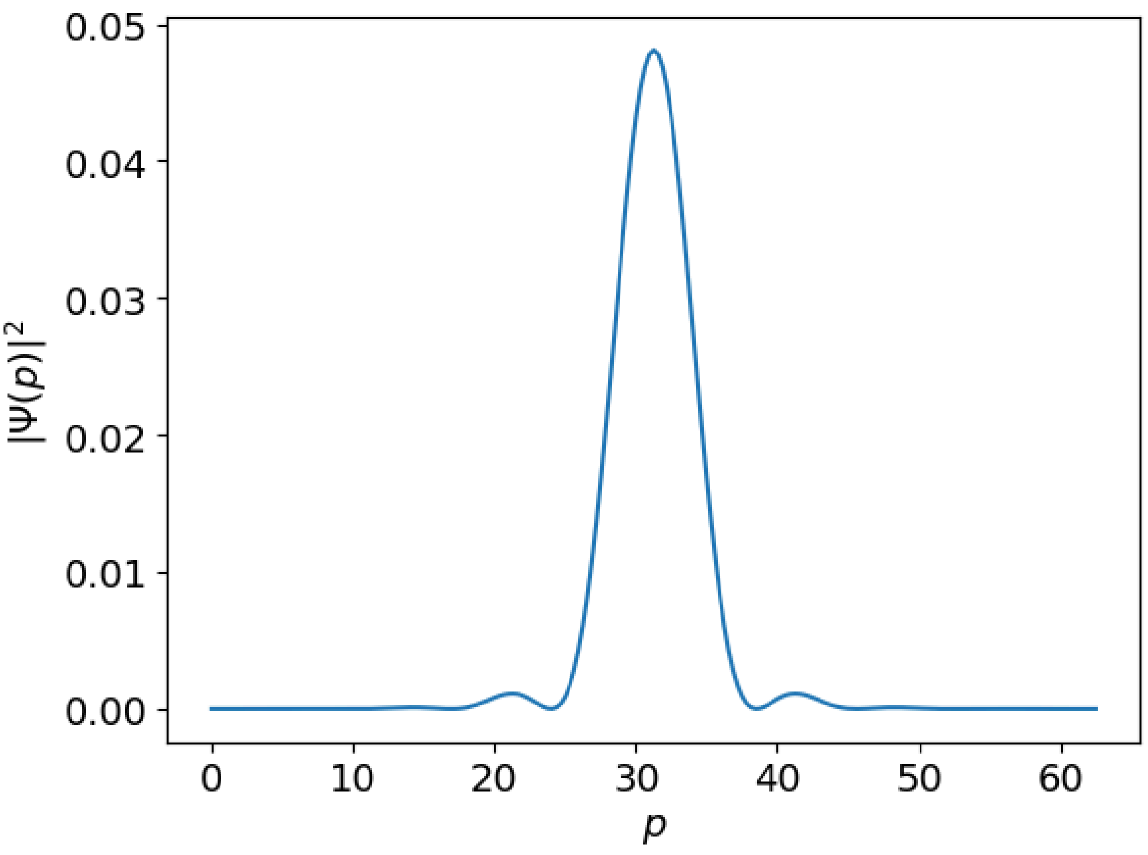
```
def MomentumBasis(L,delta_x):
    N=int(round(L/delta_x))+1
    ns=np.array(range(0,N))
    xs=np.linspace(-L/2,L/2,N,endpoint=True)
    ks=np.array(ns*2*np.pi)/(N*delta_x)
    U=1.0/np.sqrt(N)*np.exp(-1.j*np.outer(xs,ks))
    return ks,U
```

Now by getting out this unitary (`ps,U=MomentumBasis(20,0.1)`) you can then apply `U@psi` to rotate into the fourier basis. Take the wave-function in real space and rotate it into the fourer basis. Plot the wave-function squared in the momentum basis and see if (after measurement) in the momentum basis you are likely to get consistent answers for the momentum.

? Answer (start)

In [4]:
```
def MomentumBasis(L,delta_x):
    N=int(round(L/delta_x))+1
    ns=np.array(range(0,N))
    xs=np.linspace(-L/2,L/2,N,endpoint=True)
    ks=np.array(ns*2*np.pi)/(N*delta_x)
    U=1.0/np.sqrt(N)*np.exp(-1.j*np.outer(xs,ks))
    return ks,U

ks,U=MomentumBasis(L,delta_x)

psi_pb=U@psi

q=np.abs(psi_pb)**2
p=ks
plt.plot(p,q)
plt.xlabel("$p$")
plt.ylabel("$|\Psi (p)|^2$")
```

Out[4]:
```
Text(0, 0.5, '$|\\Psi (p)|^2$')
```



✓ Answer (end)

While our position resolution is good, our momentum resolution is not that great - it seems we can locate the momentum to within 5'sh.

b. Measuring Uncertainty

To get a better handle on this, we want to be able to quantify the uncertainty. The measure of certainty essentially needs to be a measure of the spread of the wave-function which can be quantified by the variance of the wavefunction in the position or momentum basis - i.e.

- $\Delta X = \sqrt{\langle X^2 \rangle - \langle X \rangle^2}$
- $\Delta P = \sqrt{\langle P^2 \rangle - \langle P \rangle^2}$

Recall that you can get the expectation value of an observable O by doing `psi.conjugate() @ O @ psi.T`

Write a function `def uncertainty(psi,X,P)` which returns the ΔX and ΔP and $\Delta X \Delta P$.

Now compute the uncertainty for the wave-functions above. You should quantify what we say above. The variance of the position is smallish but the variance of the momentum is much larger and so if we measured we would be getting widely different momentum.

? Answer (start)

In [5]:
```python
def uncertainty(psi,X,P):

    Xsqr=X @ X
    PSqr=P2
    psi_dag=psi.conjugate()

    X_exp= (psi_dag @ X @ psi.T).item()
    X2_exp= (psi_dag @ Xsqr @ psi.T).item()
    P_exp= (psi_dag @ P @ psi.T).item()
    P2_exp= (psi_dag @ PSqr @ psi.T).item()

    # strip tiny imaginary parts
    X_exp  = np.real_if_close(X_exp)
    X2_exp = np.real_if_close(X2_exp)
    P_exp  = np.real_if_close(P_exp)
    P2_exp = np.real_if_close(P2_exp)

    Delta_X2=X2_exp - X_exp**2
    Delta_P2=P2_exp - P_exp**2

    # delete small negative numerical noise

    Delta_X2 = max(0.0, np.real(Delta_X2))
    Delta_P2 = max(0.0, np.real(Delta_P2))

    Delta_X=np.sqrt(Delta_X2)
    Delta_P=np.sqrt(Delta_P2)

    return Delta_X, Delta_P, Delta_X*Delta_P
```

```
Delta_X,Delta_P, UncertaintyXP = uncertainty(psi,X,P)

print("Delta_X=", Delta_X)
print("Delta_P=", Delta_P)
print("Delta_X*Delta_P=", UncertaintyXP)
```

```
Delta_X= 0.23331078991742396
Delta_P= 2.894141767377324
Delta_X*Delta_P= 0.6752345018798129
```

✓ Answer (end)

Our wave-function does have a tuning paramater α that will make it more or less concentrated in x (for example try $\alpha = 0.0001$ in the x-basis). Write some code to tune α from 10 down to 0.0001. On the same plot, show as a function of α

- the value of ΔX
- the value of ΔP
- the value of $\Delta X \Delta P$

What is the minimum value of $\Delta X \Delta P$ you can find. The uncertainty principle tells us that $\Delta X \Delta P > 0.5$. What's the closest you get?

(you may find it useful here to work on a logarithmic grid of alpha - i.e.

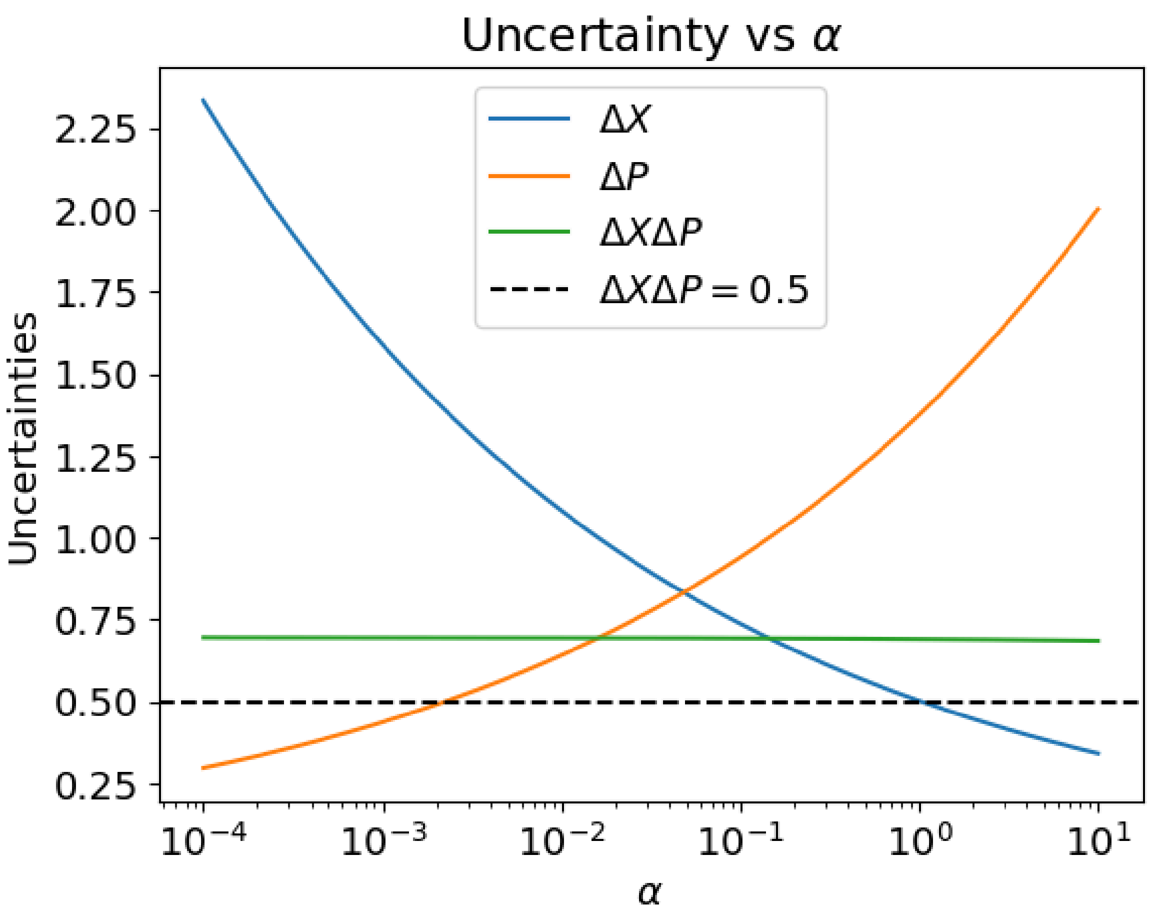
```
alphas=np.arange(0.0001,10.01,0.01)
alphas=np.exp(np.arange(-10,10,1.0))
```

If you do this, then you are going to want to change the x-axis on a log-scale as `plt.xscale('log')`

Plot also the theoretical minima `plt.ahline(0.5)`

? Answer (start)

In [6]:
```
alphas = np.logspace(-4, 1, num=200)

y_DX=np.zeros(alphas.size)
y_DP=np.zeros(alphas.size)
y_DXDP=np.zeros(alphas.size)

for i,alpha in enumerate(alphas):

    psi=waveFunction(xs,alpha)
    psi=normalize(psi)

    Delta_X,Delta_P, UncertaintyXP = uncertainty(psi,X,P)

    y_DX[i]=Delta_X
    y_DP[i]=Delta_P
    y_DXDP[i]=UncertaintyXP


plt.plot(alphas,y_DX, label="$\Delta X$")
plt.plot(alphas,y_DP,label="$\Delta P$")
plt.plot(alphas,y_DXDP, label="$\Delta X \Delta P$")
plt.axhline(0.5, color='k', linestyle='--', label=r'$\Delta X \Delta P = 0.5$')
plt.ylabel("Uncertainties")
plt.title(r'Uncertainty vs $\alpha$')
plt.xlabel("$\\alpha$")
plt.xscale('log')
plt.legend()
plt.show()

# minimum value of \Delta_X * \Delta_P

min_val = np.min(y_DXDP)
i_min   = np.argmin(y_DXDP)
alpha_min = alphas[i_min]

print("Min ΔXΔP =", min_val, "at α =", alpha_min)
```



```
Min ΔXΔP = 0.6860241470917923 at α = 10.0
```

✓ Answer (end)

c. Minimizing Uncertainty

So far we've been having trouble finding something that is compact both in real space and in momentum space. In particular, we haven't found a wave-function that matches the Heisenberg limit. In this section, we are going to try to optimize for the minimum uncertainty. (*you want to do this section with a grid spacing of* `dx=0.1`).

We will perform this optimization using python's `scipy.optimize`. To do this, we need to write an objective function `def f(psi)` that takes the wave-function `psi` and returns the thing we want to minimize.

You could do this by calling your `uncertainty` function and just returning $\Delta X \Delta P$. It will be useful to add to our objective function a small penalty for cases where either ΔX or ΔP are very large (we want to avoid this because we want to know x and p both pretty well and this helps the optimization because there is a subtle issue with our discretization as $\Delta X \to 0$ and ΔP gets very large).

Therefore, use an objective function $\Delta X \Delta P + 0.01 \max(\Delta X, \Delta P)$. Make sure your three delta are all real.

Now we can call the optimization function. The optimization function needs to be sent out objective function and a guess - i.e. `ans=scipy.optimize.minimize(f,guess)`. You can use as your guess $\sin(0.3141592653589793x)$ which you can check is not very localized.

Once the optimizations is done, the new wave-function is `ans.x`. Check the uncertainty of your new wave-function and plot it both in the x-basis and the p-basis. You may notice that your wave-function looks gaussian. You can double check this by trying to fit/plot a gaussian on top of it.

? Answer (start)

In [7]:
```python
#Objective function

def f(psi_flat):
    psi = normalize(psi_flat)
    DeltaX, DeltaP, DeltaXP = uncertainty(psi, X, P)

    DeltaX  = float(DeltaX)
    DeltaP  = float(DeltaP)
    DeltaXP = float(DeltaXP)

    return DeltaXP + 0.01 * max(DeltaX, DeltaP)
```

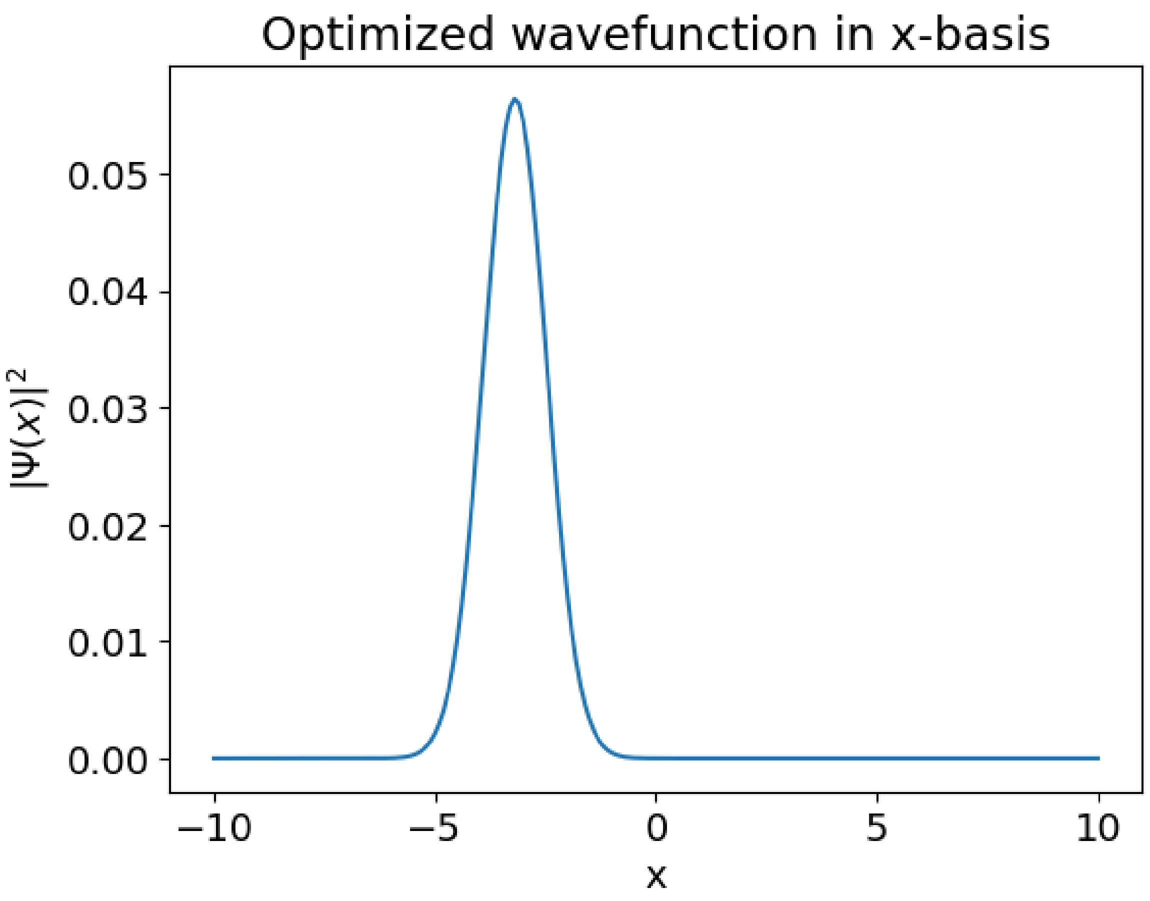
In [8]:
```python
#Guess function
guess=np.sin(0.3141592653589793 * xs)
guess=normalize(guess)

#Optimized function
ans=scipy.optimize.minimize(f, guess) #, method='BFGS')
```

In [9]:
```python
psi_opt = ans.x.astype(complex)

psi_opt = normalize(psi_opt)

# Check uncertainties for optimized state
DeltaX_opt, DeltaP_opt, DeltaXP_opt = uncertainty(psi_opt, X, P)
print("Optimized ΔX    =", DeltaX_opt)
print("Optimized ΔP    =", DeltaP_opt)
print("Optimized ΔXΔP  =", DeltaXP_opt)
```

```
Optimized ΔX     = 0.70688570764392
Optimized ΔP     = 0.7068857059728584
Optimized ΔXΔP   = 0.499687402489996
```

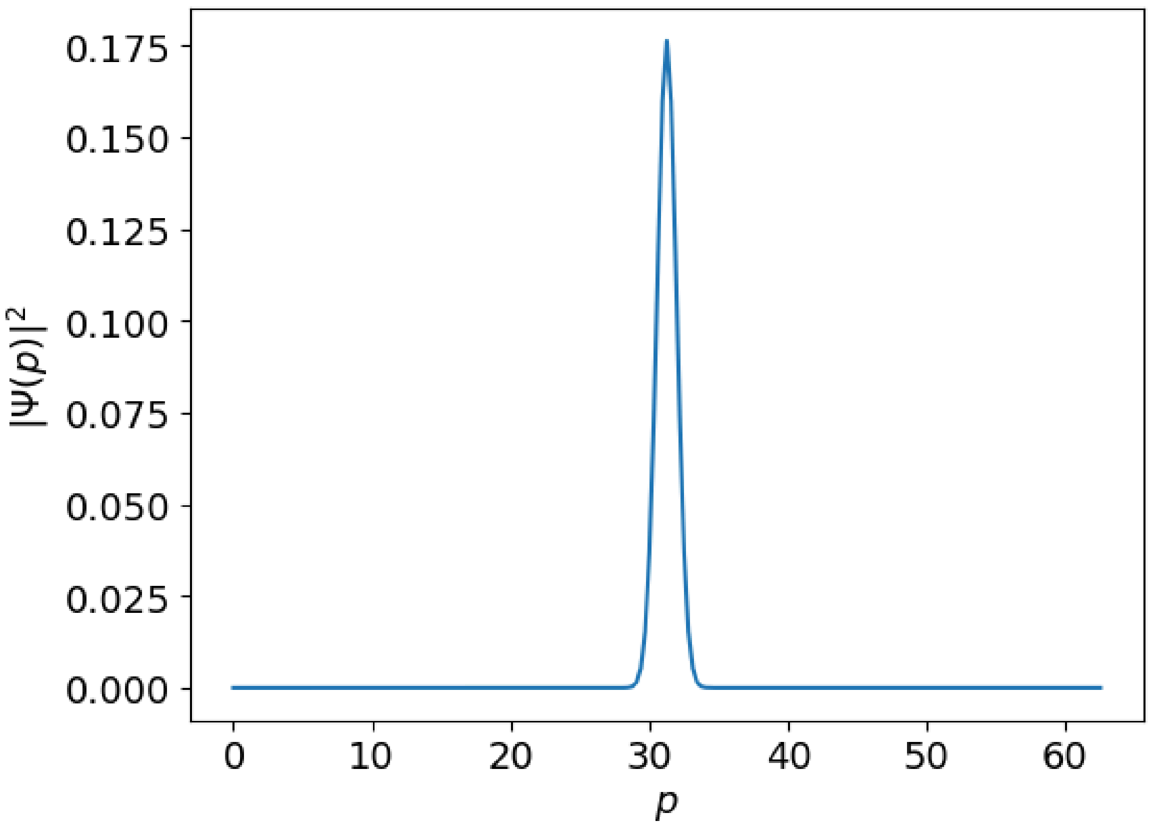
In [10]:
```
plt.figure()
plt.plot(xs, np.abs(psi_opt)**2)
#if np.max(np.abs(psi_opt.imag)) > 1e-12:
#    plt.plot(xs, psi_opt.imag, label=r'$\mathrm{Im}\,\psi(x)$', linestyle='--')
plt.xlabel("x")
plt.ylabel("$|\Psi (x)|^2$")
plt.title("Optimized wavefunction in x-basis")
```

Out[10]:
```
Text(0.5, 1.0, 'Optimized wavefunction in x-basis')
```



In [11]:
```
psi_pb=U@psi_opt

q=np.abs(psi_pb)**2
p=ks
plt.plot(p,q)
plt.xlabel("$p$")
plt.ylabel("$|\Psi (p)|^2$")
```

Out[11]:
```
Text(0, 0.5, '$|\\Psi (p)|^2$')
```



✓ Answer (end)

d. Coherent States

There is a special wave-function that we will consider later called a coherent state.

The wave-function for coherent states are

$$\Psi(x) = e^{-\frac{1}{2}(x-\alpha\sqrt{2})^2}$$

We will set $\alpha = 0.2$ for this part of the problem.

Using your code, compute the uncertainty ΔX and ΔP and Δ for this (actually will be true for any) coherent state?

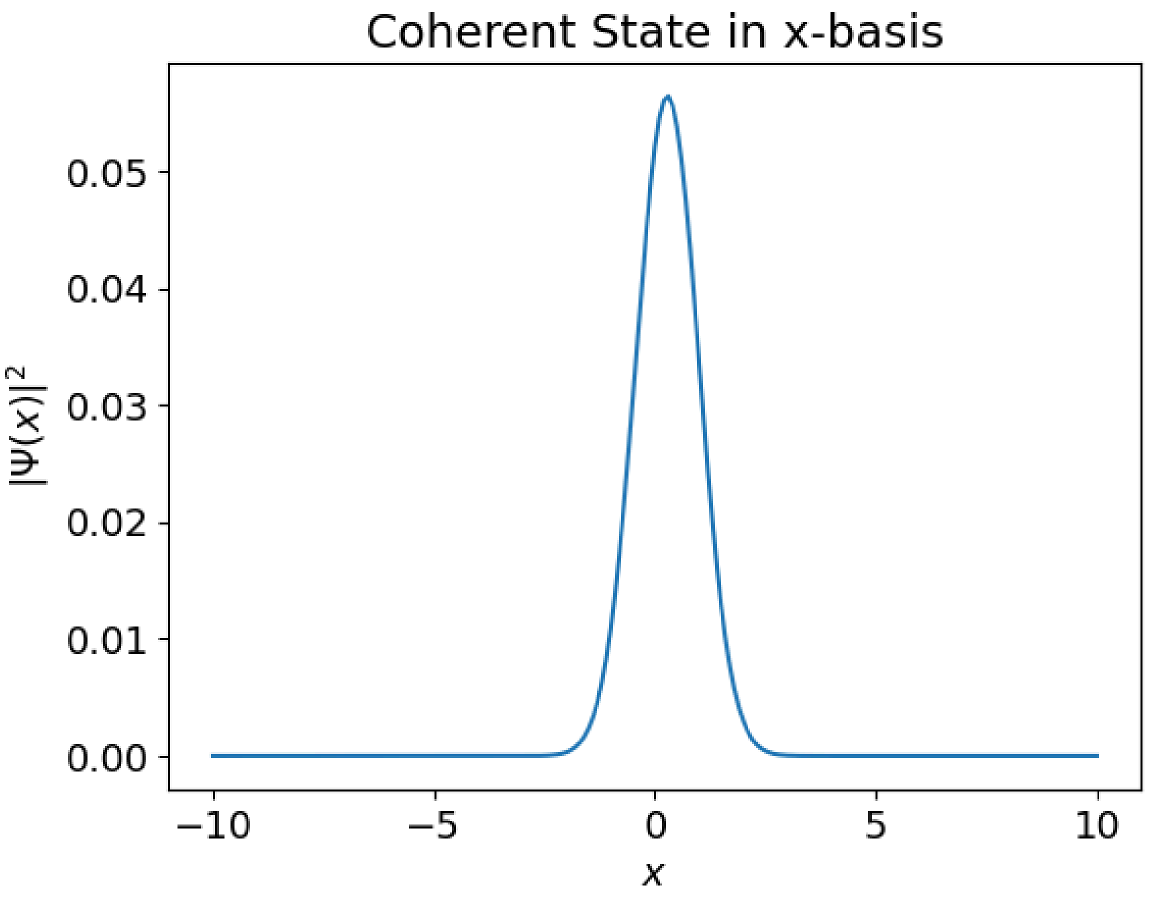
Also plot the probability in real and momentum space.

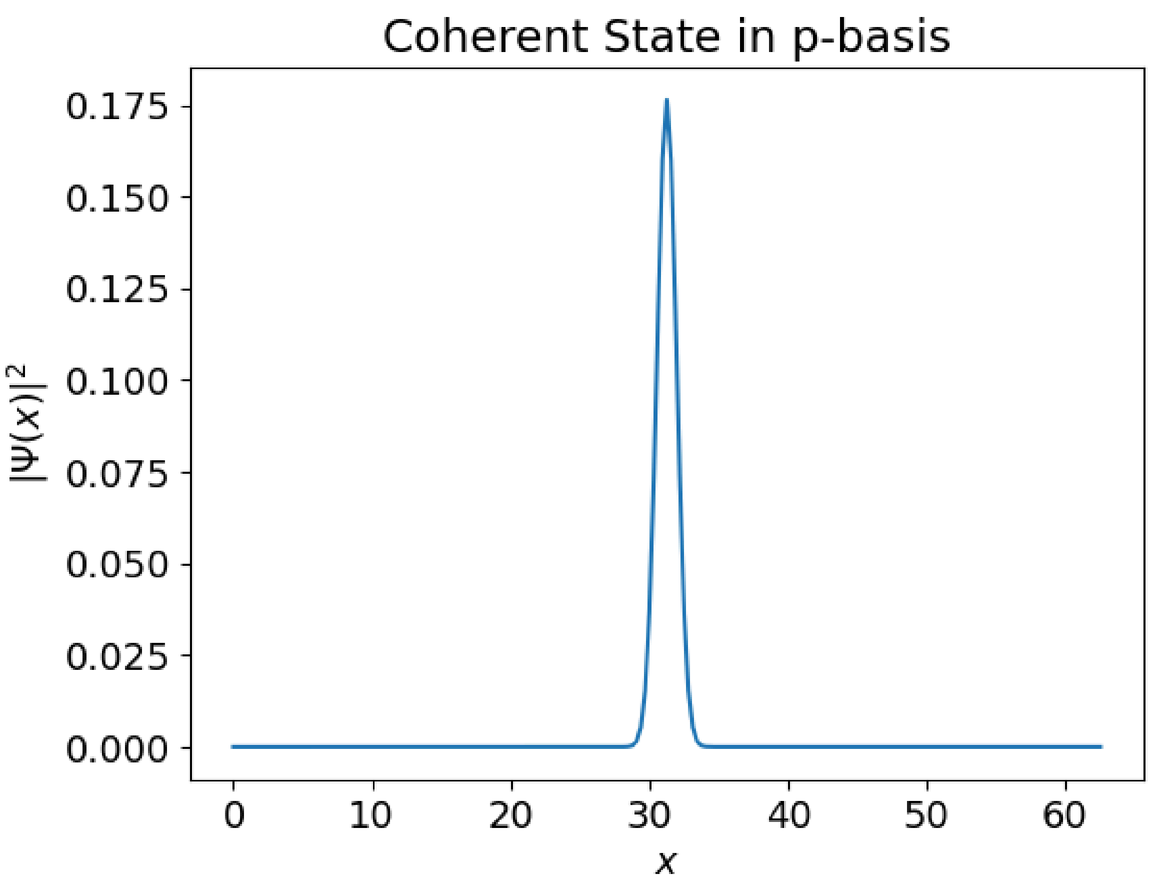
? Answer (start)

In [12]:
```
#Defining the coherent state
def coherentState(x, alpha=0.2):
    return np.exp(-(1/2.0)*(x-alpha*np.sqrt(2))**2)

psi=coherentState(xs)
psi=normalize(psi)

DeltaX,DeltaP, DeltaXP = uncertainty(psi,X,P)
print("ΔX     =", DeltaX)
print("ΔP     =", DeltaP)
print("ΔXΔP   =", DeltaXP)
```

```
plotPDfunction(psi,xs)
plt.title("Coherent State in x-basis")
plt.show()

ps,U=MomentumBasis(L,delta_x)
psi_pb=U@psi
plotPDfunction(psi_pb,ps)
plt.title("Coherent State in p-basis")
plt.show()
```

```
ΔX     = 0.7071067811865475
ΔP     = 0.7066650695400019
ΔXΔP   = 0.49968766269939846
```





✓ Answer (end)

e. Not coherent states

So far we've seen examples of states which satiate the uncertainty bound while simultaneously having the same uncertainty in x and p. Verify that this state satiates the Heisenberg uncertainty bound while having different values of ΔX and ΔP

$$\Psi(x) = N \exp[-0.3x^2]$$

where N is the normalization.

? Answer (start)

In [13]:
```
def nonCoherentState(x, alpha=0.3):
    return np.exp(-alpha*(x**2))

psi=nonCoherentState(xs)
psi=normalize(psi)

DeltaX,DeltaP, DeltaXP = uncertainty(psi,X,P)
print("ΔX     =", DeltaX)
print("ΔP     =", DeltaP)
print("ΔXΔP   =", DeltaXP)
```

```
ΔX     = 0.9128709291752769
ΔP     = 0.5475172257179016
ΔXΔP   = 0.49981255858057066
```

✓ Answer (end)

Exercise 2. Harmonic Oscillator in Real Space

The Hamiltonian for the simple harmonic oscillator is

$$H = \frac{\hat{P}^2}{2m} + \frac{1}{2}m\omega^2\hat{X}^2$$

We know that $\hat{P} = -i\hbar\frac{\partial}{\partial x}$ leaving us with

$$H = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}m\omega^2x^2$$

Working in units where $\hbar = 1$ and selecting $m = \omega = 1$ we are then left with

$$H = -\frac{1}{2}\frac{\partial^2}{\partial x^2} + \frac{1}{2}x^2$$

a. Hamiltonian and Eigenvalues

Let's start by generating the matrix for this Hamiltonian. We are again going to use $-10 \leq x \leq 10$ via

```
delta_x=0.01
xs=SetupGrid(20,delta_x)
X,P,P2=SetupObservables(xs,delta_x)
```

To generate the Hamiltonian

- The Kinetic piece is as it was in the particle in the box
- The Potential piece is diagonal with $V(x) = 1/2x^2$ down the diagonal.

Plotting the first 10 eigenenergies. You should start trying to make your plots pretty and informative:

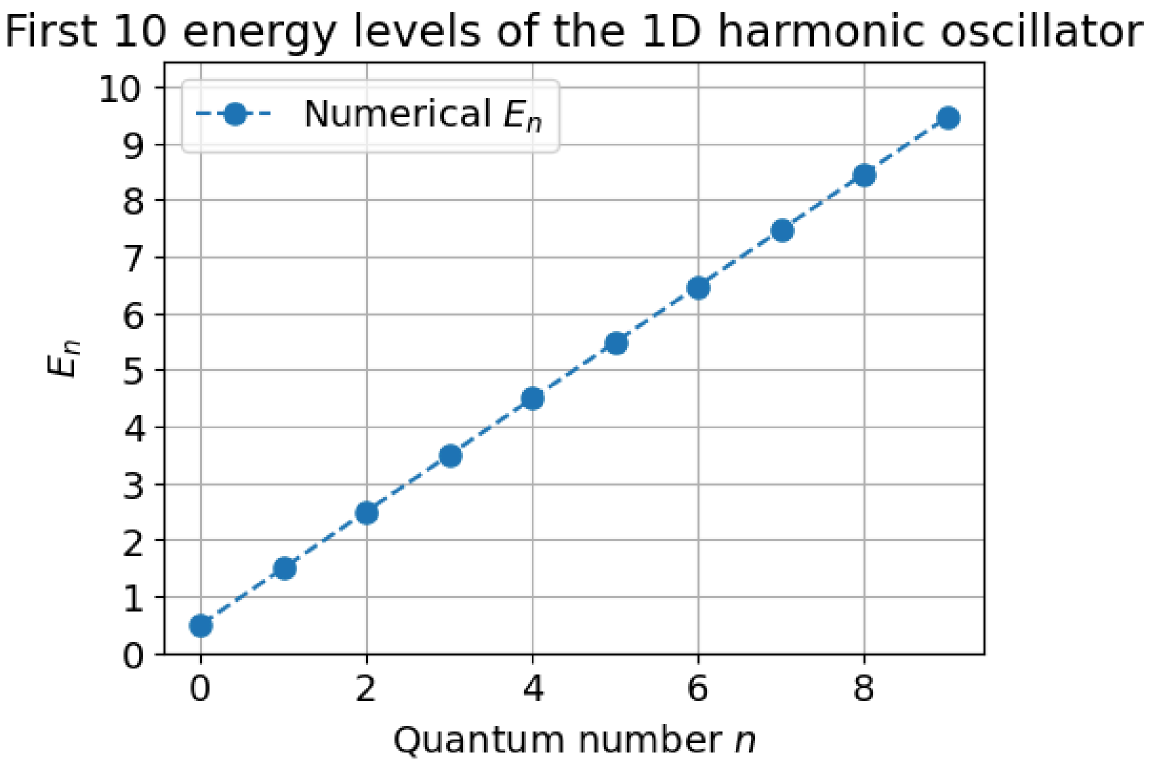
- It's often useful to set up the size of the plot by doing `fig,ax=plt.subplots(1,1,figsize=(5, 4),constrained_layout=True)`
- Include in your plots reasonable markers and sizes `marker='o',markersize=10,linestyle='--'`
- Turn on a grid (`ax.grid()`)
- Make the grid have a reasonable step `ax.yaxis.set_major_locator(plt.MultipleLocator(1))`
- Label your axes! (`ax.set_xlabel(...)`)
- Set your limits to something sane (`ax.set_ylim(0,something)`)

What important qualitative feature do you notice about the energy levels?

? Answer (start)

In [14]:
```python
xs = SetupGrid(20, delta_x)
X, P, P2 = SetupObservables(xs, delta_x)

T = 0.5 * P2                        # kinetic term (assuming P2 ≈ P@P)
V_diag = 0.5 * xs**2                # V(x) = x^2 / 2
V = np.diag(V_diag)                 # diagonal potential matrix

H = T + V

# Find eigenvalues (energies) and eigenvectors (stationary states)
E, v = np.linalg.eigh(H)

# first 10 eigenvalues
n_levels = 10
E10 = E[:n_levels]
n = np.arange(n_levels)  # 0,1,...,9 (these correspond to n=0..9 states)
```

In [15]:
```python
fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=True)

ax.plot(n, E10, marker='o', markersize=8, linestyle='--', label='Numerical $E_n$
ax.set_xlabel("Quantum number $n$")
ax.set_ylabel("$E_n$")
ax.set_title("First 10 energy levels of the 1D harmonic oscillator")
ax.grid(True)
# make y-axis ticks every 1 unit
ax.yaxis.set_major_locator(plt.MultipleLocator(1))
# Set a sane energy range
ax.set_ylim(0, E10[-1] + 1)
ax.legend()
plt.show()
```



✓ Answer (end)

b. Eigenstates

Now we want to plot the lowest three eigenstates. You should add labels to your states: in each of your plot commands do

- `ax.plot(x,y,label="Eigenstate ...")` and then include the legend with
- `ax.legend()`

Notice how many nodes (zeros) the n'th eigenstate has.

In class, you learned that the expected solution to the Harmonic Oscillator involves Hermite polynomials. In particular, the n'th eigenstate should be

$$\Psi_n(x) = \frac{1}{\sqrt{2^n n!}}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right)$$

where H_n are the Hermite polynomials.

In python to get the Hermite polynomials, use

```
Herm.hermval(xi, herm_coeffs)
```

where $xi = \sqrt{m\omega/\hbar}$ and the `herm_coefs` need to be a numpy array of size n+1 with a 1 in spot n and 0 otherwise.

- Plot the expected answers on top of your generated answers. You should find that they fall on top of each other.

You may occassionally find that the expected and numerical answer are upside down from each other. This is because the global phase of a wave-function doesn't matter. This means you can always multiply a wave-function by -1 and it's still physically the same. If you find this is happening to you, go ahead and multipy appropriately by -1 to fix it.

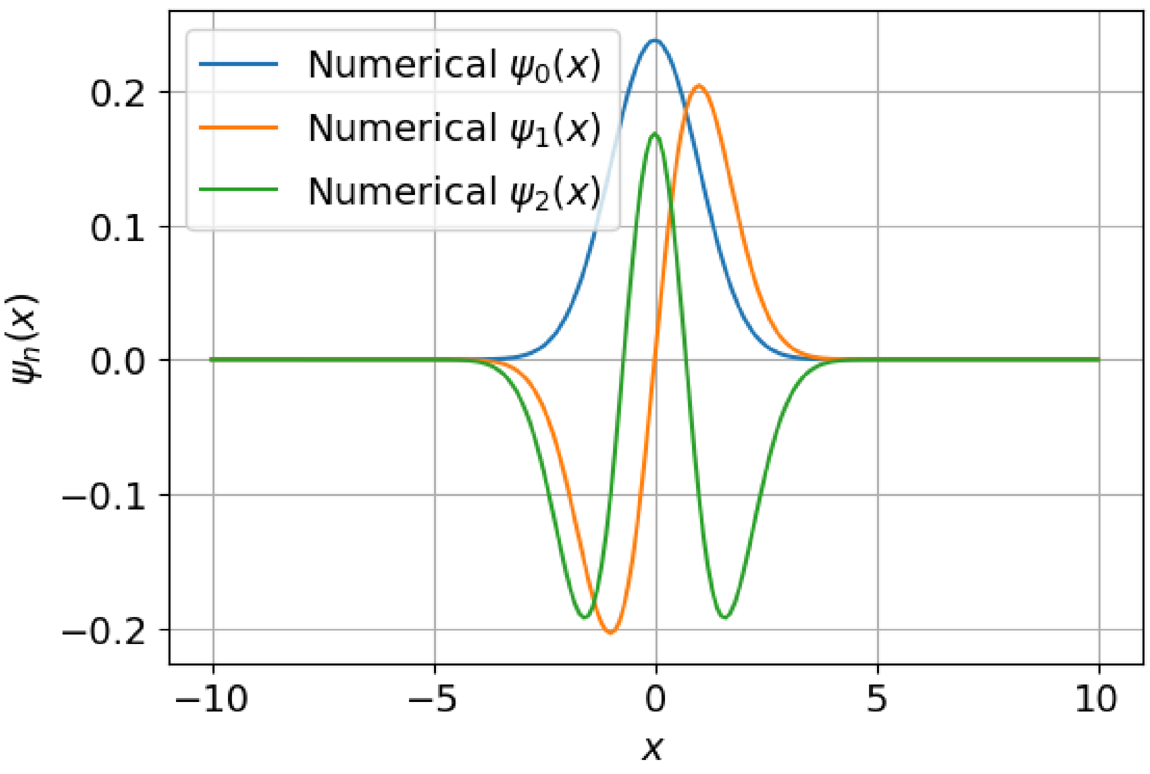
- In addition, print out the uncertainty of these wave-functions. Are they anywhere near the Heisenberg limit.

? Answer (start)

In [16]:
```
# numerical stationary states
psi0num = normalize(v[:, 0])
psi1num = normalize(v[:, 1])
psi2num = normalize(v[:, 2])

fig, ax = plt.subplots(1, 1, figsize=(6, 4), constrained_layout=True)

ax.plot(xs, psi0num.real, label="Numerical $\psi_0(x)$")
ax.plot(xs, psi1num.real, label="Numerical $\psi_1(x)$")
ax.plot(xs, psi2num.real, label="Numerical $\psi_2(x)$")

ax.set_xlabel("$x$")
ax.set_ylabel("$\psi_n(x)$")
#ax.set_title("Lowest three numerical HO eigenstates")
```

```
ax.grid(True)
ax.legend()
```

Out[16]:
```
<matplotlib.legend.Legend at 0x12ccae279d0>
```



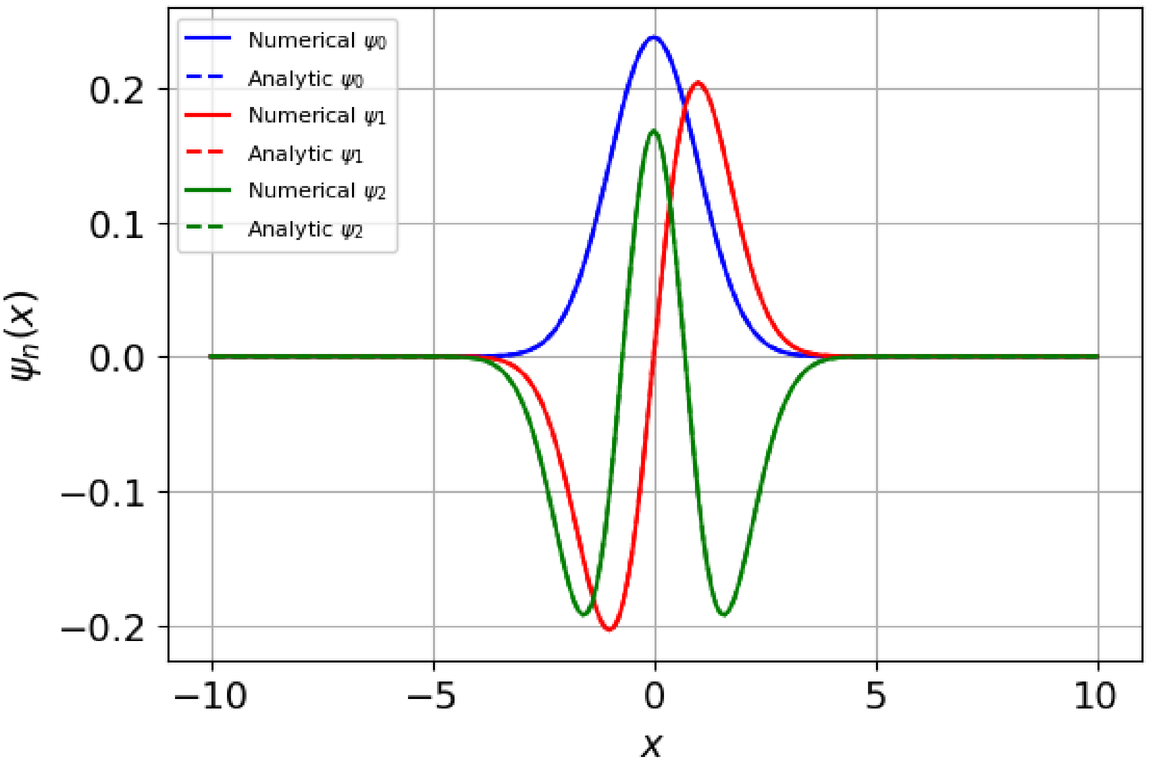
In [17]:
```
def HO_analytic_statState(xs, n):
    # Recall that we're using m=ω=ħ=1
    # Hermite polynomial H_n(x)
    coeffs = np.zeros(n+1)
    coeffs[n] = 1.0
    Hn = Herm.hermval(xs, coeffs)

    # Normalization factor
    NormFac = (1.0/np.pi)**0.25 / np.sqrt(2.0**n * np.math.factorial(n))

    psi = NormFac * np.exp(-0.5*xs**2) * Hn
    # Normalizing in the discrete grid
    psi = normalize(psi)
    return psi
```

In [18]:
```
psi0theo = HO_analytic_statState(xs, 0)
psi1theo = HO_analytic_statState(xs, 1)
psi2theo = HO_analytic_statState(xs, 2)
```

```
C:\Users\ariel\AppData\Local\Temp\ipykernel_12936\2838844749.py:9: DeprecationWar
ning: `np.math` is a deprecated alias for the standard library `math` module (Dep
recated Numpy 1.25). Replace usages of `np.math` with `math`
  NormFac = (1.0/np.pi)**0.25 / np.sqrt(2.0**n * np.math.factorial(n))
```

In [20]:
```
# Define function to align phases

def align_phase(psi_num, psi_th):
    # ensure both are column vectors
    psi_num = psi_num.reshape(-1,1)
    psi_th  = psi_th.reshape(-1,1)
    overlap = np.vdot(psi_num, psi_th)  # To check whether the phase is the same
    if overlap.real < 0:
        psi_th = -psi_th
    return psi_th

psi0theo = align_phase(psi0num, psi0theo)
psi1theo = align_phase(psi1num, psi1theo)
psi2theo = align_phase(psi2num, psi2theo)
```

In [25]:
```
#Plotting
fig, ax = plt.subplots(1, 1, figsize=(6, 4), constrained_layout=True)

ax.plot(xs, psi0num.real, 'b-',  label="Numerical $\psi_0$")
ax.plot(xs, psi0theo.real,  'b--', label="Analytic $\psi_0$")

ax.plot(xs, psi1num.real, 'r-',  label="Numerical $\psi_1$")
ax.plot(xs, psi1theo.real,  'r--', label="Analytic $\psi_1$")

ax.plot(xs, psi2num.real, 'g-',  label="Numerical $\psi_2$")
ax.plot(xs, psi2theo.real,  'g--', label="Analytic $\psi_2$")

ax.set_xlabel("$x$")
ax.set_ylabel("$\psi_n(x)$")
ax.grid(True)
ax.legend()
ax.legend(loc='upper left', fontsize=8)
plt.show()
plt.close()
```



In [24]:
```
for n, psi in enumerate([psi0num, psi1num, psi2num]):
    Delta_X, Delta_P, Delta_XP = uncertainty(psi, X, P)
    print(f"n = {n}: ΔX = {Delta_X:.4f}, ΔP = {Delta_P:.4f}, ΔXΔP = {Delta_XP:.4
```

```
n = 0: ΔX = 0.7067, ΔP = 0.7071, ΔXΔP = 0.4997
n = 1: ΔX = 1.2235, ΔP = 1.2247, ΔXΔP = 1.4984
n = 2: ΔX = 1.5786, ΔP = 1.5811, ΔXΔP = 2.4959
```

✓ Answer (end)

c. Dynamics and Ehrenfest theorem

Like we did in the particle in the box, we are going to do some dynamics.

It's worth encapsulating your time-evolution into two functions (*if you've done the particle-in-a-finite-well assignment you may have already written these functions*)

- `TimeEvolutionOperator(H,delta_t)` should take the Hamiltonian and your time step and return $e^{-iH\delta t}$. It's important to note that for each Hamiltonian, you only need to compute this time-evolution operator once even if you are time-evolving for many steps (or even many different initial wave-functions). This will save significant time compared to computing it each time!

- `TimeEvolution(psi,M,steps)` which takes the initial wave-function `psi`, the time-evolution operator `M` and the number of steps and returns a list of arrays that include the snapshot of the wave-function at every time step.

We also want to do an update of the update function so that the animation is slightly more informative. *Note: If you did the particle-in-a-finite-well assignment you may have already done these modifications* In particular, we are going to want to plot some information about the potential and energy of our state. Because our y-axis is currently probability (which is at most 1), we will need to rescale the energy (which can get large) so that everything sanely fits on the same axis. To do this, we are going to rescale all our energies by `scale=max_value/energy` where `max_value` is the largest probability your wave-function gets (which we've already been calculating for animation) and `energy` is the energy of our initial wave-function, which you can compute (recall the energy doesn't change as a function of time).

Modifications of the Update Function

- Add a line which takes the `energy` and `max_value` sent to it and computes a scale
- Plots a red-dashed line at the energy of our state - i.e. `plt.axhline(energy*scale,color='red',linestyle='--')`
- Plots the potential (scaled by the scale)
- Gives `def update(frame, skip, xs, positions, potential, max_value, energy)` the relevant additional parameters. To do this you will need to compute the positions using your `X` from `SetupObservables` (outside the function)

Let's start with the wave-function

- `psi=(np.sqrt(0.1)*v[:,0]+np.sqrt(0.25)*v[:,1]+1.j*np.sqrt(0.65)*v[:,2]).ast` where `v` are the eigenstates from the SHO.

Perform time-evolution with the initial wave-function with a

- $\delta t = 0.1$ and
- for a total time of $T = 40$.

Store the snapshot of the wave-function at each time and animate it.

? Answer (start)

In [ ]:
```
# Time evolution functions

def TimeEvolutionOperator(H, delta_t):
    H = np.asarray(H)
    M = scipy.linalg.expm(-1j * H * delta_t)
    return M

def TimeEvolution(psi0, M, steps):
    # Ensure column vector
    psi = np.asarray(psi0, dtype=complex).reshape(-1, 1)
    snapshots = [psi.copy()]   # include t=0

    for _ in range(steps):
        psi = M @ psi          # apply time evolution
        snapshots.append(psi.copy())

    return snapshots
```

In [ ]:
```
# Time evolution set-up
delta_t = 0.1
T = 40.0
steps = int(T / delta_t)
M = TimeEvolutionOperator(H, delta_t)
```

In [ ]:
```
# Initial wave function
psi = (np.sqrt(0.1)*v[:,0] + np.sqrt(0.25)*v[:,1] + 1.j*np.sqrt(0.65)*v[:,2])#.c
psi=normalize(psi)

# Cast the wave function into a numpy column vector
psi_t0=np.asarray(psi, dtype=complex).reshape(-1, 1)
```

In [ ]:
```
# Generating the snapshots
myData = TimeEvolution(psi_t0, M, steps)      #List of size steps + 1
```

In [ ]:
```
#To save the time evolution computed in the previous block use:
np.save("TimeDepWaveHO.npy", myData)
```

In [ ]:
```
myData = np.load("TimeDepWaveHO.npy")

# Modified Update function
def update(frame, skip, xs, positions, potential, max_value, energy):
    # Clear current axes
    plt.cla()
    # Actual index in myData
    idx = frame * skip
    psi_t = np.asarray(myData[idx])
    prob  = np.abs(psi_t)**2
    # Rescaling factor so energy & potential fit on same y-axis as probability
    scale = max_value / energy
    # Plot probability density |ψ(x,t)|^2
    plt.plot(xs, prob, label=r'$|\Psi(x,t)|^2$')
    # Plot scaled potential V(x)*scale
    plt.plot(xs, potential * scale, color='black', linestyle=':', label='V(x)')
    # Plot scaled energy E*scale as red dashed line
    plt.axhline(energy * scale, color='red', linestyle='--', label='Energy (scal
    # Plot the expectation value ⟨X⟩(t) as a marker
    plt.scatter([positions[idx]], [0], color='green', s=30, label=r'$\langle X \

    # Axis limits and labels
    plt.ylim(0, max_value * 1.1)
    plt.xlabel('x')
    plt.ylabel('Probability / scaled energy')
    plt.title(f'Frame {idx+1}/{len(myData)}')
    plt.legend(loc='upper right', fontsize=8)
```

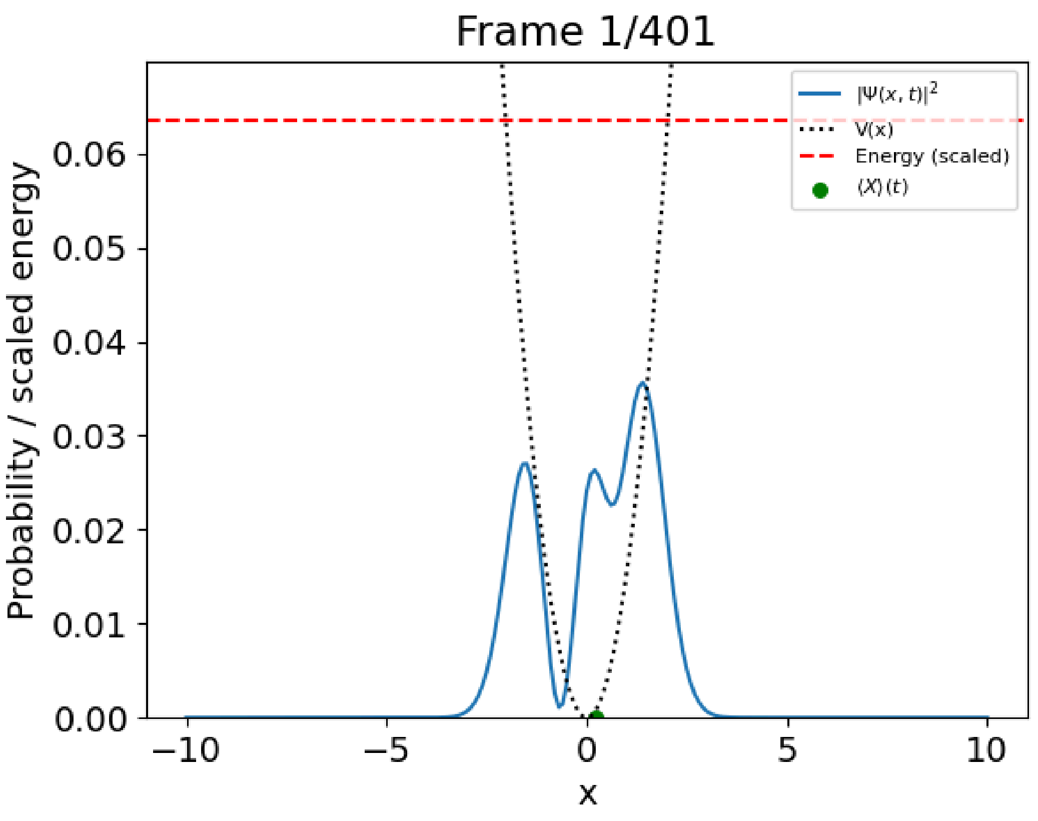
In [ ]:

```
# Computing the data that will be passed to update
n_steps = len(myData)
# ⟨X⟩(t) for each time step
positions = np.zeros(n_steps)
for i, psi in enumerate(myData):
    psi_vec = np.asarray(psi).reshape(-1, 1)
    positions[i] = np.real((psi_vec.conjugate().T @ X @ psi_vec).item())

# Compute the energy of the initial state, recall that energy is constant in tim
energy = np.real((psi_t0.conjugate().T @ H @ psi_t0).item())

# Compute global max probability (for y-axis scaling)
max_value = 0.0
for psi in myData:
    prob = np.abs(psi)**2
    max_value = max(max_value, np.max(prob))

# Define the potential V(x) = 1/2 x^2 for the HO
potential = 0.5 * xs**2

#Frames skiped and number of frames
skip = 1   # draw every 5th time step
n_frames = n_steps // skip
```

In [ ]:

```
# Create a figure and axis
fig, ax = plt.subplots()
#   Animation
animation = FuncAnimation(
    fig,
    update,
    frames=n_frames,
    fargs=(skip, xs, positions, potential, max_value, energy),
    interval=40,   # ms between frames
    repeat=True
)
display(HTML(animation.to_jshtml()))
plt.close()
```

```
Animation size has reached 20981763 bytes, exceeding the limit of 20971520.0. If
you're sure you want a larger animation embedded, set the animation.embed_limit r
c parameter to a larger value (in MB). This and further frames will be dropped.
```



✓ Answer (end)

As in the particle in the box, takes your list of wave-functions as a function of time and generate from each wave-function a value of $\langle x\rangle(t)$ and $\langle p\rangle(t)$ showing (on the subplots as typical)

On the same plot show

- $\langle x\rangle(t)$
- $\langle p\rangle(t)$
- $\partial\langle x\rangle(t)/\partial t$

What do you notice about the second two plots?

We are also going to add one more subplot this time: `ax=plt.subplots(3,1)[1]`

For our third subplot what we'd like to plot is

- $\frac{\partial\langle p(t)\rangle}{\partial t}$ generated by finite differences or `np.gradient` using your momentum curve above

- -$\langle V'(x)\rangle(t)$ This is the average of the derivative of the potential over the wave-function. Take the derivative of the potential by hand and then think about what observable you need to get this.

You should find these latter two bullet points are on top of each other.

This (and the previous plots) are a validation of Ehrenfest Theorem which tells us

$$m\frac{d}{dt}\langle x\rangle = \langle p\rangle$$

and

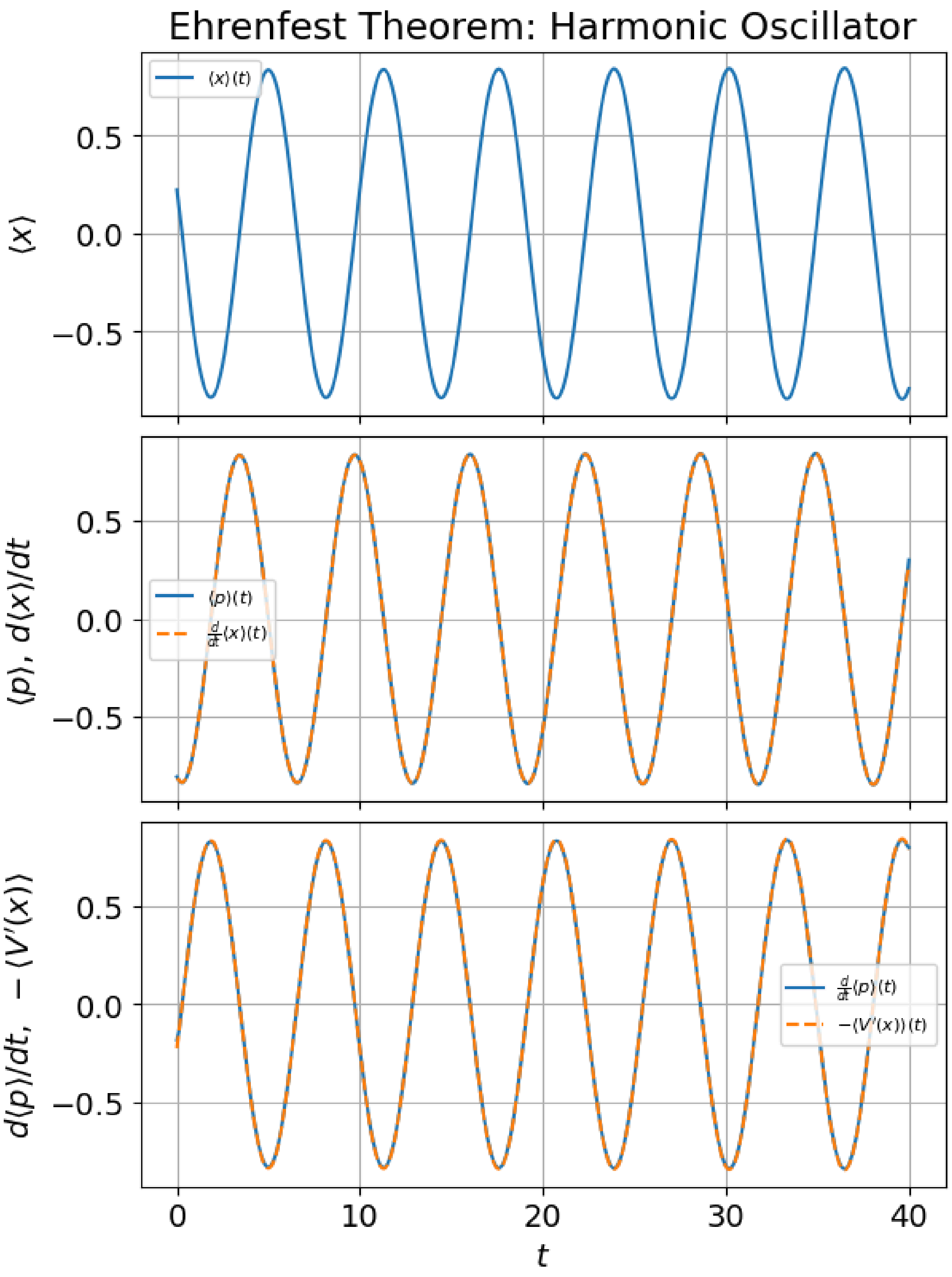
$$\frac{d}{dt}\langle p\rangle = -\langle V'(x)\rangle$$

Finally go ahead and make a phase plot of momentum vs position.

? Answer (start)

In [ ]:
```python
# Computing ⟨X⟩(t) and ⟨P⟩(t)

n_steps = len(myData)
times = np.arange(n_steps) * delta_t

x_expect = np.zeros(n_steps)
p_expect = np.zeros(n_steps)

for i, psi_t in enumerate(myData):
    psi = np.asarray(psi_t, dtype=complex).reshape(-1, 1)
    psi_dag = psi.conjugate().T

    x_expect[i] = np.real((psi_dag @ X @ psi).item())
    p_expect[i] = np.real((psi_dag @ P @ psi).item())

# Computing time derivatives d⟨X⟩/dt and d⟨P⟩/dt

dx_dt = np.gradient(x_expect, delta_t)
dp_dt = np.gradient(p_expect, delta_t)

# Computing -⟨V'(x)⟩(t) = -⟨(1/2)(x^2)'⟩(t) = -⟨x⟩(t)

minus_Vprime_expect = -x_expect


# Plots
fig, axes = plt.subplots(3, 1, figsize=(6, 8), constrained_layout=True, sharex=T
ax1, ax2, ax3 = axes

# Subplot 1: <x>(t)
ax1.plot(times, x_expect, label=r'$\langle x \rangle(t)$')
ax1.set_ylabel(r'$\langle x \rangle$')
ax1.set_title(r'Ehrenfest Theorem: Harmonic Oscillator')
ax1.grid(True)
ax1.legend(loc='best', fontsize=8)

# Subplot 2: <p>(t) and d<x>/dt
ax2.plot(times, p_expect, label=r'$\langle p \rangle(t)$')
ax2.plot(times, dx_dt, linestyle='--', label=r'$\frac{d}{dt}\langle x \rangle(t)
ax2.set_ylabel(r'$\langle p \rangle,\, d\langle x\rangle/dt$')
ax2.grid(True)
ax2.legend(loc='best', fontsize=8)
```

```
# Subplot 3: d<p>/dt and -<V'(x)>
ax3.plot(times, dp_dt, label=r'$\frac{d}{dt}\langle p \rangle(t)$')
ax3.plot(times, minus_Vprime_expect, linestyle='--', label=r'$-\langle V^\prime(
ax3.set_xlabel(r'$t$')
ax3.set_ylabel(r'$d\langle p\rangle/dt,\ -\langle V^\prime(x)\rangle$')
ax3.grid(True)
ax3.legend(loc='best', fontsize=8)

plt.show()
```



What do we notice in the second two plots?

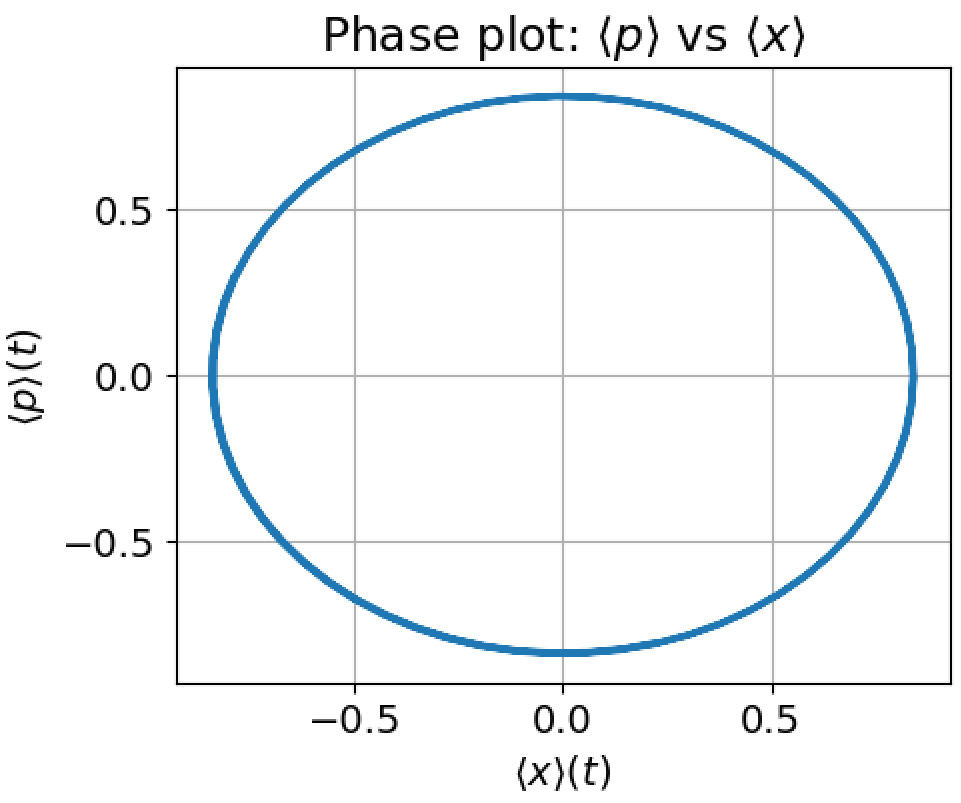
- Subplot 2: Both curves lie on top of each other because $m\frac{d}{dt}\langle x\rangle = \langle p\rangle$, and m=1, implies that:

$$\frac{d}{dt}\langle x\rangle = \langle p\rangle$$

- Subplot 3: Both curves also lie on top of each other because

$$\frac{d}{dt}\langle p\rangle = -\langle V(x)\rangle$$

In [ ]:
```
# phase plot

plt.figure(figsize=(5, 4))
plt.plot(x_expect, p_expect)
plt.xlabel(r'$\langle x \rangle(t)$')
plt.ylabel(r'$\langle p \rangle(t)$')
plt.title('Phase plot: $\langle p \\rangle$ vs $\langle x \\rangle$')
plt.grid(True)
plt.show()
```



This looks like an ellipse, as we expect from the classical motion in the x–p plane.

✓ Answer (end)

d. Coherent States

We now wish to run time dynamics on a coherent state. Recall the coherent state (as above) is

$$\Psi(x) = \frac{1}{N} e^{-\frac{1}{2}(x-\alpha\sqrt{2})^2}$$

where N normalizes the wave-function. We will use $\alpha = 3$

Do time-dynamics on this system generating both the animation as well as the other plots above.

Add one more line to your plots: the position and velocity of a classical oscillator in the same Harmonic potential starting at $x = 3\sqrt{2}$

What do you notice that's interesting about the time-evolution of the coherent states.

Does something not quite look right in your animation?

? Answer (start)

In [ ]:
```
#Defining the coherent state
alpha = 3.0
def coherent_state(xs, alpha):
    psi = np.exp(-0.5 * (xs - alpha*np.sqrt(2))**2)
    psi = normalize(psi)
    return psi

psi0 = coherent_state(xs, alpha)

#Time evolution
delta_t = 0.1
T       = 40.0
steps   = int(T / delta_t)   # 400

M = TimeEvolutionOperator(H, delta_t)
myDataCS = TimeEvolution(psi0, M, steps)  # list of ψ(t), length steps+1
```

In [ ]:
```
#To save the time evolution computed in the previous block use:
np.save("TimeDepWaveHO_CoheStates.npy", myDataCS)
```

In [ ]:
```
myDataCS = np.load("TimeDepWaveHO_CoheStates.npy")

#Expectation values of x, p and derivatives

n_steps = len(myDataCS)
times   = np.arange(n_steps) * delta_t

x_expect = np.zeros(n_steps)
p_expect = np.zeros(n_steps)

for i, psi_t in enumerate(myDataCS):
    psi= np.asarray(psi_t, dtype=complex).reshape(-1, 1)
    psi_dag = psi.conjugate().T

    x_expect[i] = np.real((psi_dag @ X @ psi).item())
    p_expect[i] = np.real((psi_dag @ P @ psi).item())

dx_dt = np.gradient(x_expect, delta_t)
dp_dt = np.gradient(p_expect, delta_t)
```

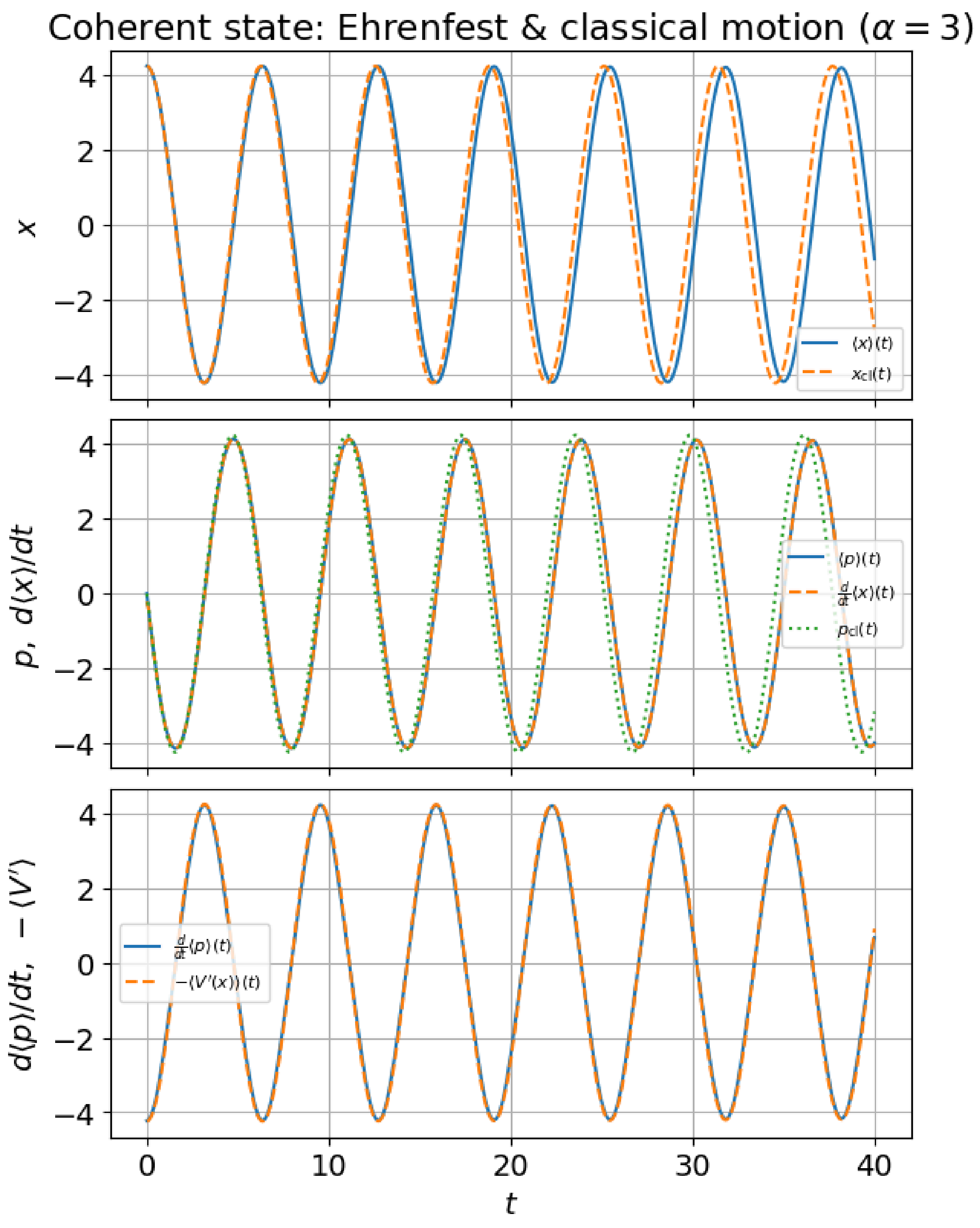
For a classical oscillator with $m = \omega = 1$ and initial conditions:

- $x(0) = 3\sqrt(2)$
- $p(0) = 0$

the solution is

$$x_{cl} = 3\sqrt{2}\cos(t) \quad p_{cl} = -3\sqrt{2}\sin(t)$$

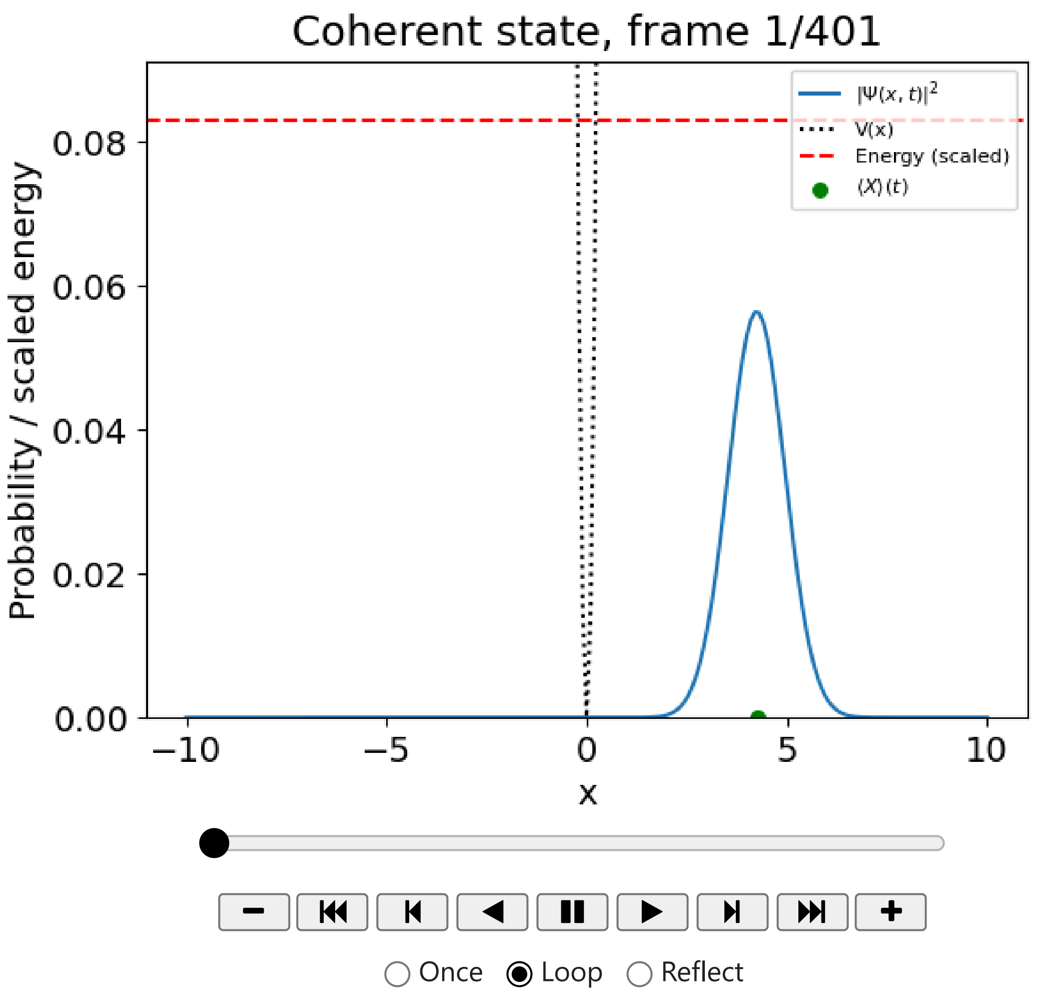
In [ ]:
```
#Classical Harmonic oscillator with same initial values
A = np.sqrt(2)*alpha
x_cl = A * np.cos(times)
p_cl = -A * np.sin(times)

#Subplots compared to classical harmonic oscillator
fig, axes = plt.subplots(3, 1, figsize=(6, 8),
                         constrained_layout=True, sharex=True)
ax1, ax2, ax3 = axes

#Subplot 1: <x>(t) and classical x(t)
ax1.plot(times, x_expect, label=r'$\langle x \rangle(t)$')
ax1.plot(times, x_cl,    '--', label=r'$x_{\rm cl}(t)$')
ax1.set_ylabel(r'$x$')
ax1.set_title(r'Coherent state: Ehrenfest & classical motion ($\alpha=3$)')
ax1.grid(True)
ax1.legend(loc='best', fontsize=8)

#Subplot 2: <p>(t), d<x>/dt and classical p(t)
ax2.plot(times, p_expect, label=r'$\langle p \rangle(t)$')
ax2.plot(times, dx_dt,    '--', label=r'$\frac{d}{dt}\langle x\rangle(t)$')
ax2.plot(times, p_cl,     ':', label=r'$p_{\rm cl}(t)$')
ax2.set_ylabel(r'$p,\ d\langle x\rangle/dt$')
ax2.grid(True)
ax2.legend(loc='best', fontsize=8)

#Subplot 3: d<p>/dt and -<V'(x)> (for HO, V'(x)=x)
minus_Vprime_expect = -x_expect  # since V'(x)=x for HO

ax3.plot(times, dp_dt, label=r'$\frac{d}{dt}\langle p \rangle(t)$')
ax3.plot(times, minus_Vprime_expect, '--', label=r'$-\langle V^\prime(x)\rangle(
ax3.set_xlabel(r'$t$')
ax3.set_ylabel(r'$d\langle p\rangle/dt,\ -\langle V^\prime\rangle$')
ax3.grid(True)
ax3.legend(loc='best', fontsize=8)

plt.show()
```



```
In [ ]: # Computing the data that will be passed to update
        n_steps = len(myDataCS)
        # ⟨X⟩(t) for each time step
        positions = x_expect

        # Compute the energy of the initial state, recall that energy is constant in tim
        energy = np.real((psi0.conjugate().T @ H @ psi_t0).item())

        # Compute global max probability (for y-axis scaling)
        max_value = 0.0
        for psi in myDataCS:
            prob = np.abs(psi)**2
            max_value = max(max_value, np.max(prob))

        # Define the potential V(x) = 1/2 x^2 for the HO
        potential = 0.5 * xs**2
```

```
#Frames skiped and number of frames
skip = 1   # draw every 5th time step
n_frames = n_steps // skip

def update(frame, skip, xs, positions, potential, max_value, energy):
    # Clear current axes
    plt.cla()
    # Actual index in myDataCS
    idx = frame * skip
    psi_t = np.asarray(myDataCS[idx])
    prob  = np.abs(psi_t)**2
    # Rescaling factor so energy & potential fit on same y-axis as probability
    scale = max_value / energy
    # Plot probability density |ψ(x,t)|^2
    plt.plot(xs, prob, label=r'$|\Psi(x,t)|^2$')
    # Plot scaled potential V(x)*scale
    plt.plot(xs, potential * scale, color='black', linestyle=':', label='V(x)')
    # Plot scaled energy E*scale as red dashed line
    plt.axhline(energy * scale, color='red', linestyle='--', label='Energy (scal
    # Plot the expectation value ⟨X⟩(t) as a marker
    plt.scatter([positions[idx]], [0], color='green', s=30, label=r'$\langle X \

    # Axis limits and labels
    plt.ylim(0, max_value * 1.1)
    plt.xlabel('x')
    plt.ylabel('Probability / scaled energy')
    plt.title(f'Coherent state, frame {frame+1}/{len(myDataCS)}')
    plt.legend(loc='upper right', fontsize=8)

# Create a figure and axis
fig, ax = plt.subplots()
#   Animation
animation = FuncAnimation(
    fig,
    update,
    frames=n_frames,
    fargs=(skip, xs, positions, potential, max_value, energy),
    interval=40,   # ms between frames
    repeat=True
)
display(HTML(animation.to_jshtml()))
plt.close()
```

```
Animation size has reached 20996047 bytes, exceeding the limit of 20971520.0. If
you're sure you want a larger animation embedded, set the animation.embed_limit r
c parameter to a larger value (in MB). This and further frames will be dropped.
```



What do you notice that's interesting about the time-evolution of the coherent states?

- A coherent state is a minimum-uncertainty wavepacket.
- As it evolves in the harmonic oscillator, it keeps (almost) the same Gaussian shape over time.
- Its expectation values ⟨x⟩(t) and ⟨p⟩(t) follow the same trajectory as a classical oscillator with matching initial conditions:
  - $x_{\text{cl}}(t) = 3\sqrt{2}\cos t$
  - $p_{\text{cl}}(t) = -3\sqrt{2}\sin t$
- The uncertainty product ΔX·ΔP stays close to the Heisenberg minimum and barely changes.
- This shows that coherent states behave like classical particles with a quantum "fuzz".

What “doesn’t look quite right” in the animation?

- The wavepacket breathes slightly (its width oscillates a bit).
- Over long times, the trajectory deviates from the perfect classical motion.
- These imperfections arise from numerical limitations:
  - finite spatial grid (–10 to 10 only),
  - discretization errors from finite-difference P^2,
  - finite time-step Δt and matrix exponential approximations.

So, although the coherent state behaves approximately as theory predicts, the animation is not perfectly rigid or perfectly Gaussian due to numerical effects.

✓ Answer (end)

Exercise 3. Raising and Lowering Operators

a. Raising and Lowering Operators

With the Harmonic oscillator, we've learned that it is useful to look at raising and lowering operators.

$$a = \sqrt{\frac{m\omega}{2\hbar}}\left(\hat{X} + \frac{i}{mw}\hat{P}\right)$$

$$a^\dagger = \sqrt{\frac{m\omega}{2\hbar}}\left(\hat{X} - \frac{i}{mw}\hat{P}\right)$$

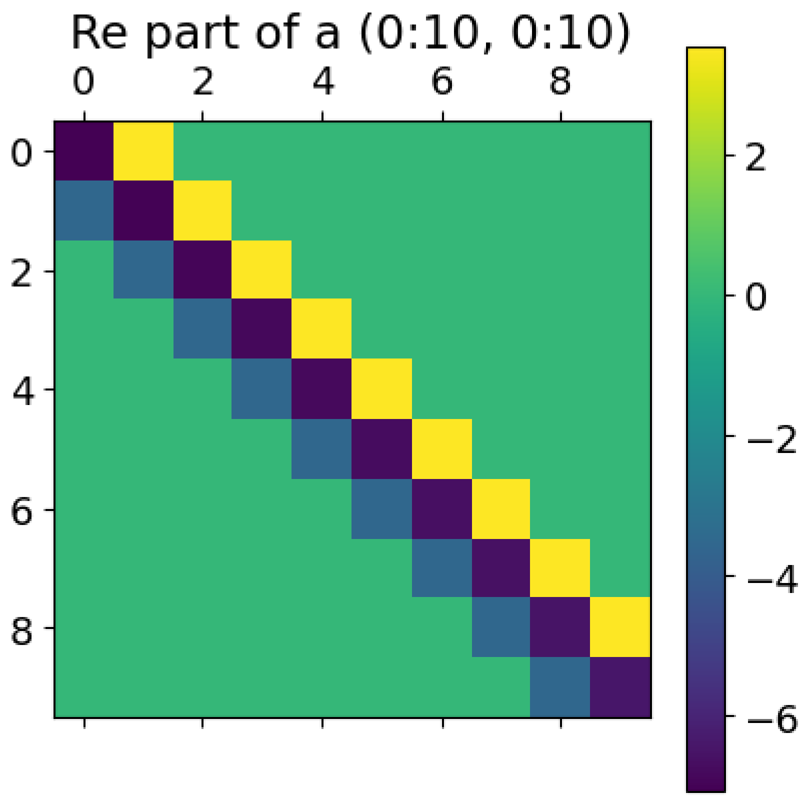
Build both $a^\dagger$ and a in the real space basis.

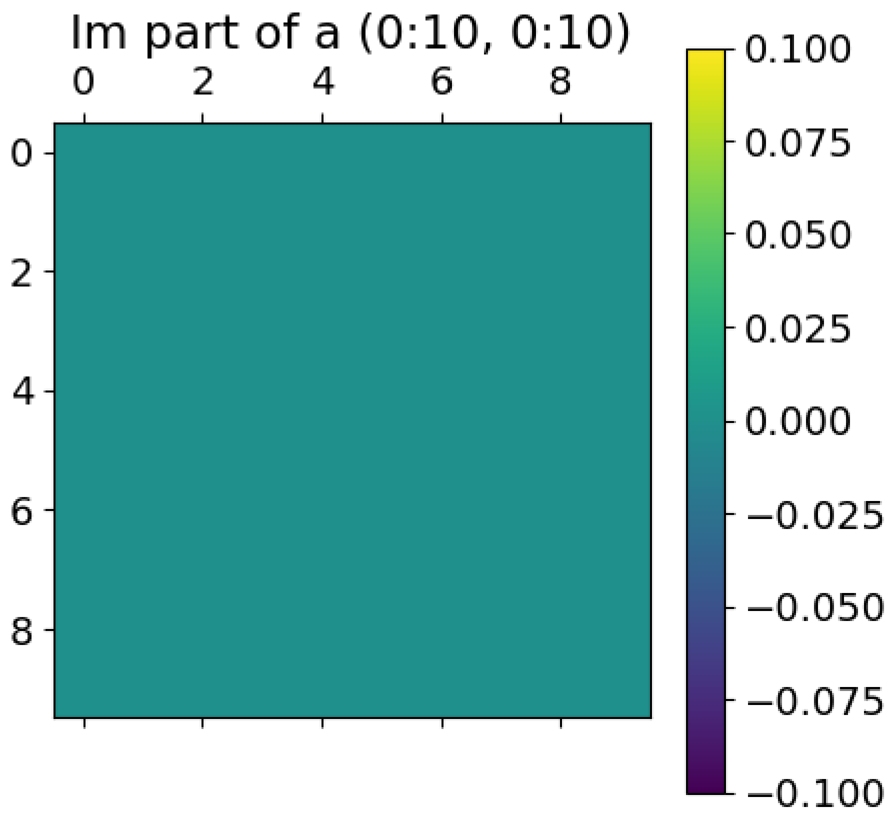
- Verify that the operators are complex conjugates of each other (by daggering a and subtracting $a^\dagger$ and making sure they are zero - i.e. check that the maximum of the absolute value of the difference is zero)
- Look at the matshow of the upper left of the matrix `plt.matshow(a[0:10,0:10])`. Can you understand why this is what this looks like.

? Answer (start)

In [ ]:
```
#Recall that X,P are hermitian operators
a     = (X + 1j * P) / np.sqrt(2)
adag  = (X - 1j * P) / np.sqrt(2)

#Checking that the operators are the complex conjugates of each other
a_dagger_from_a = a.conjugate().T
diff = a_dagger_from_a - adag

max_diff = np.max(np.abs(diff))
print("max |a†(from a) - adag| =", max_diff)

#Matshow of a
plt.matshow(a[0:10, 0:10].real)
plt.colorbar()
plt.title("Re part of a (0:10, 0:10)")
plt.show()

plt.matshow(a[0:10, 0:10].imag)
plt.colorbar()
plt.title("Im part of a (0:10, 0:10)")
plt.show()
```

```
max |a†(from a) - adag| = 0.0
```





Why does the upper-left block of the matrix `a` look the way it does?

In the real-space basis:

- The position operator X is diagonal:

$$X_{ij} = x_i \delta_{ij}$$

so it contributes only to the main diagonal of a.
- The momentum operator P is like a tridiagonal because it comes from a finite-difference derivative.
  It has nonzero values only on the bands right above and below the diagonal, and these entries are mostly imaginary.

- In the lowering operator:

$$a = \frac{1}{\sqrt{2}}(X + iP)$$

  the factor of i turns the imaginary off-diagonal structure of P into real off-diagonal elements.

Because of this:

- the diagonal of `a` comes from X,
- the two bands immediately above and below the diagonal come from iP,
- everything else is zero.

This is why `plt.matshow(a[0:10,0:10])` shows a matrix that is mostly diagonal with narrow off-diagonal bands: it is the visual combination of a diagonal position operator and a tridiagonal momentum operator.

✓ Answer (end)

b. Testing the raising of the raising operator

We expect that the raising operator has the property that when it acts on an eigenstate $|n-1\rangle$ it should produce the eigenstate $\sqrt{n}$ times eigenstate $|n\rangle$. Let's check that by plotting. Plot $\sqrt{4}|\Psi_4\rangle$ and $a^\dagger|\Psi_3\rangle$ and verify that they give the same result. Again be cognizant that you might need to flip one of the states.

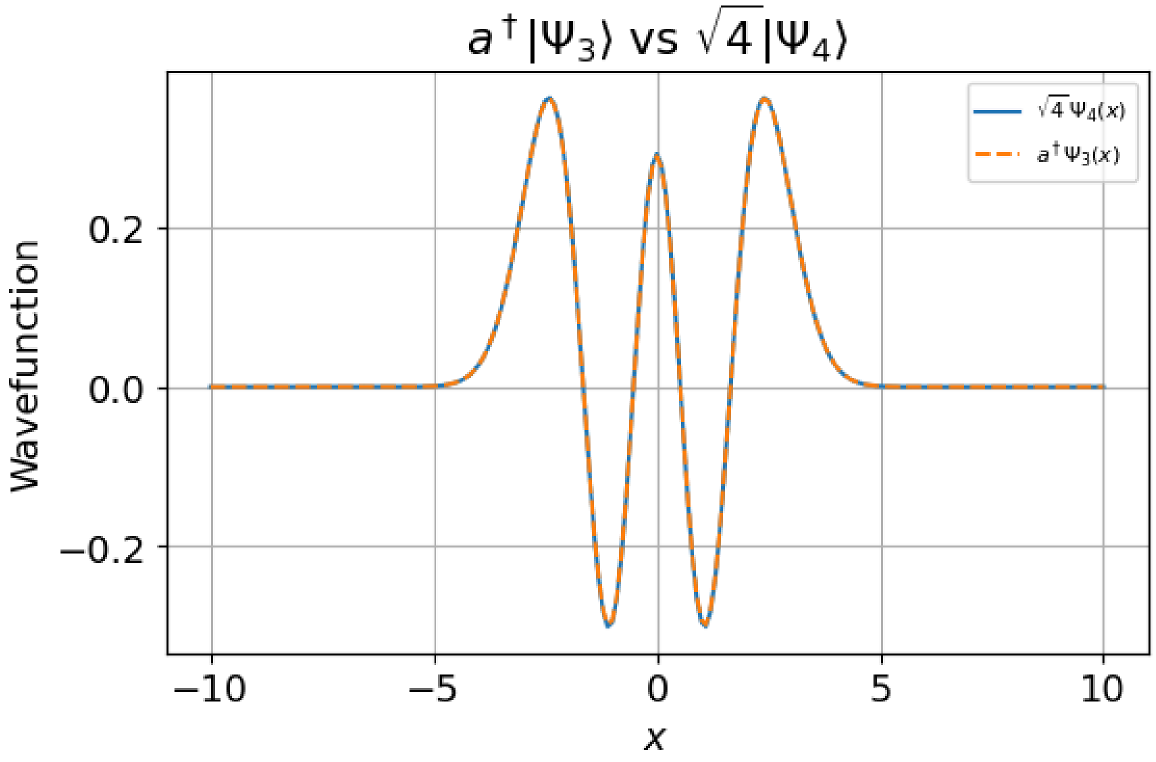
Similarly let's check that a lowers a state. Verify that eigenstate 4 is the same as eigenstate 5 lowered with a coefficient of $\sqrt{5}$.

Also verify that the lowering operator destroys the ground state - i.e. check that $a^\dagger\Psi_0$ is actually zero. (Remember to check the scale of your graph as it might have some features but essentially be zero)

? Answer (start)

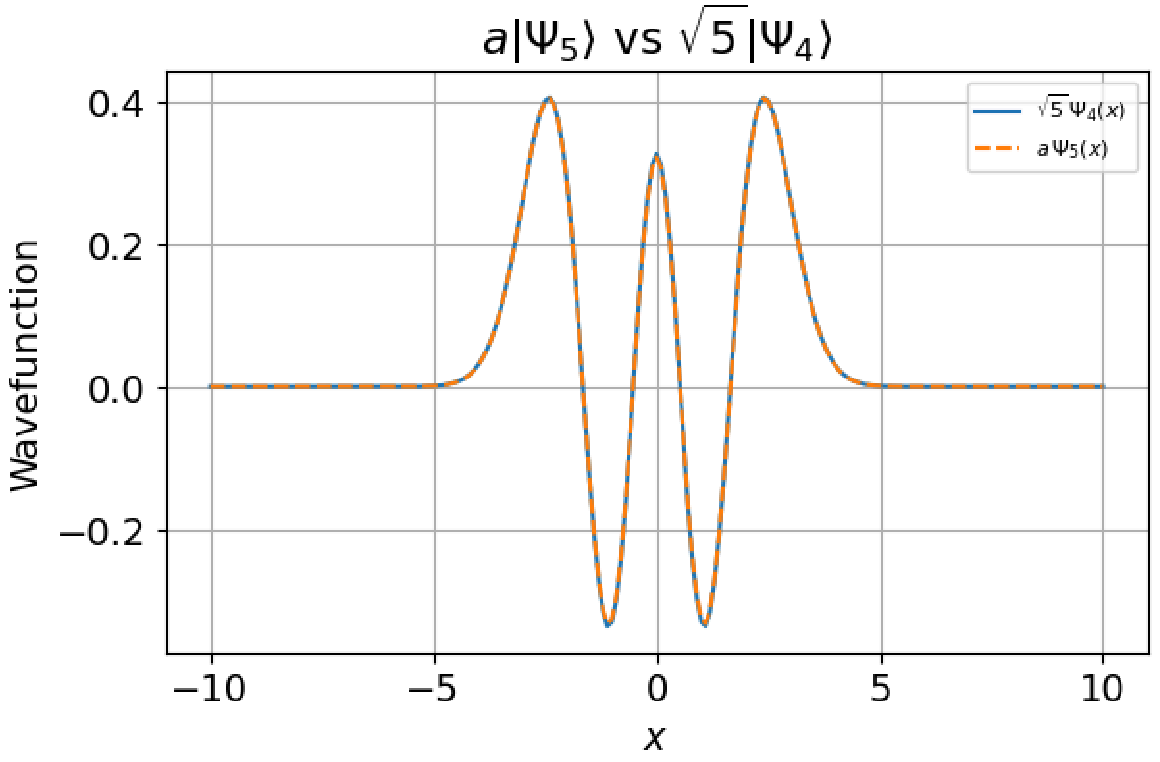
In [ ]:
```
#Define useful functions

def get_state(n):
    #Return normalized eigenstate |Psi_n> as column vector (N,1)
    psi=normalize(v[:, n])
    return psi.reshape(-1, 1)

def align_phase(psi_target, psi_to_adjust):
    #Multiply psi_to_adjust by ±1 so that its overlap with psi_target is positiv
    #This fixes the arbitrary global sign.
    psi_target = psi_target.reshape(-1, 1)
    psi_to_adjust = psi_to_adjust.reshape(-1, 1)
    overlap = np.vdot(psi_target, psi_to_adjust)
    if overlap.real < 0:
        psi_to_adjust = -psi_to_adjust
    return psi_to_adjust
```

In [ ]:
```
#Raising operator check √4 |4> = a† |3>

psi3 = get_state(3)
psi4 = get_state(4)

lhs = np.sqrt(4) * psi4          # √4 |4>
rhs = adag @ psi3                # a† |3>

# match signs
rhs_aligned = align_phase(lhs, rhs)

# plot comparison
fig, ax = plt.subplots(1, 1, figsize=(6,4), constrained_layout=True)

ax.plot(xs, lhs.real,  label=r'$\sqrt{4}\,\Psi_4(x)$')
ax.plot(xs, rhs_aligned.real, '--', label=r'$a^\dagger \Psi_3(x)$')

ax.set_xlabel('$x$')
ax.set_ylabel('Wavefunction')
ax.set_title('$a^\dagger|\Psi_3\\rangle$ vs $\\sqrt{4}|\Psi_4\\rangle$')
ax.grid(True)
ax.legend(loc='best', fontsize=8)
plt.show()

# norm of the difference
diff_norm = np.linalg.norm(lhs - rhs_aligned)
print("‖√4|4> - a†|3>‖ =", diff_norm)
```



```
‖√4|4> - a†|3>‖ = 0.012609818336149319
```

In [ ]:
```python
#Lowering operator Check √5 |4> = a |5>

psi4 = get_state(4)
psi5 = get_state(5)

lhs = np.sqrt(5) * psi4      # √5 |4>
rhs = a @ psi5               # a |5>

rhs_aligned = align_phase(lhs, rhs)

fig, ax = plt.subplots(1, 1, figsize=(6,4), constrained_layout=True)

ax.plot(xs, lhs.real,  label=r'$\sqrt{5}\,\Psi_4(x)$')
ax.plot(xs, rhs_aligned.real, '--', label=r'$a\,\Psi_5(x)$')

ax.set_xlabel('$x$')
ax.set_ylabel('Wavefunction')
ax.set_title('$a|\Psi_5\\rangle$ vs $\\sqrt{5}|\Psi_4\\rangle$')
ax.grid(True)
ax.legend(loc='best', fontsize=8)
plt.show()

diff_norm = np.linalg.norm(lhs - rhs_aligned)
print("‖√5|4> - a|5>‖ =", diff_norm)
```

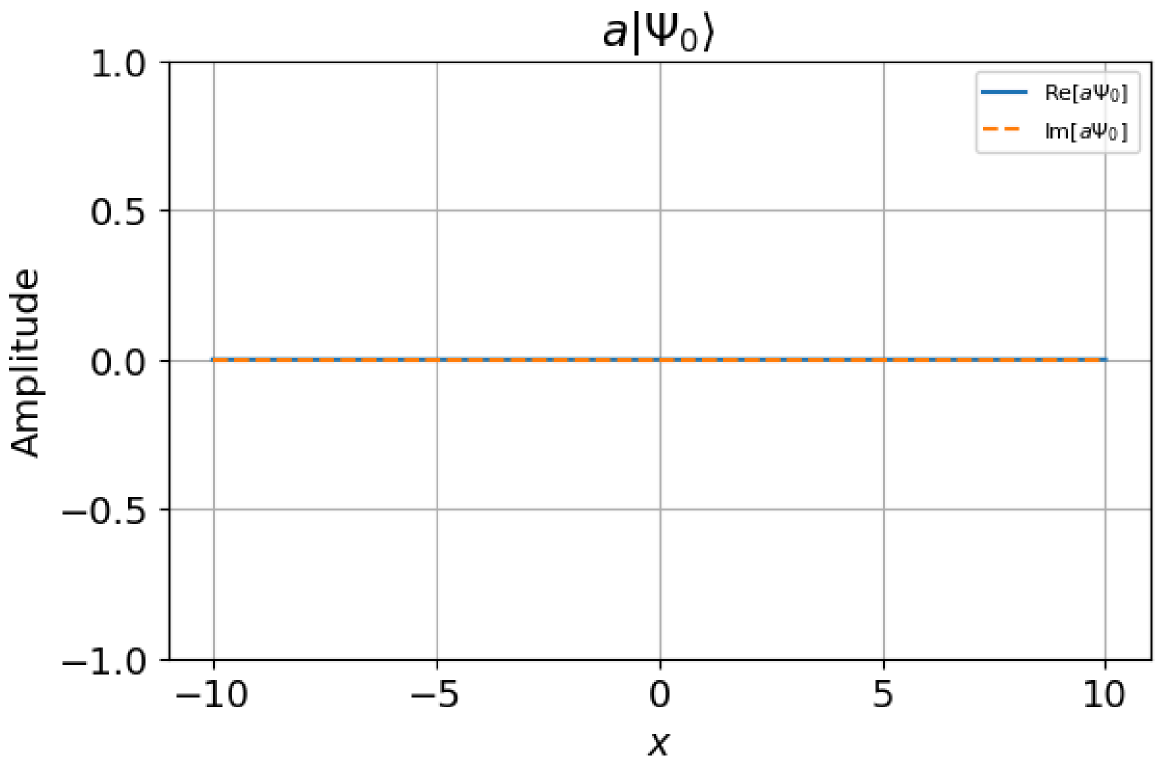


```
‖√5|4> - a|5>‖ = 0.018381810865549547
```

In [ ]:
```python
#Check that the lowering operator anihilates the groun dstate
psi0 = get_state(0)

res = a @ psi0     # a |0>
res_norm = np.linalg.norm(res)

print("‖a|0>‖ =", res_norm)
```

```
fig, ax = plt.subplots(1,1, figsize=(6,4), constrained_layout=True)
ax.plot(xs, res.real, label=r'$\mathrm{Re}[a\Psi_0]$')
ax.plot(xs, res.imag, '--', label=r'$\mathrm{Im}[a\Psi_0]$')
ax.set_xlabel('$x$')
ax.set_ylabel('Amplitude')
ax.set_title(r'$a|\Psi_0\rangle$')
ax.grid(True)
ax.set_ylim(-1, 1)
ax.legend(loc='best', fontsize=8)
plt.show()
```

```
‖a|0>‖ = 0.0009880880697321888
```



✓ Answer (end)

c. Verifying the norms and number operators

Now we would like to further verify that the raising and lowering operators always act with the additional $\sqrt{N}$ term. Since our eigenstates start normalized, if we compute the norm (`np.linalg.norm`) of the eigenstates after they've been hit by the raising and lowering operators you should find that their norm is $\sqrt{N}$. Verify this by

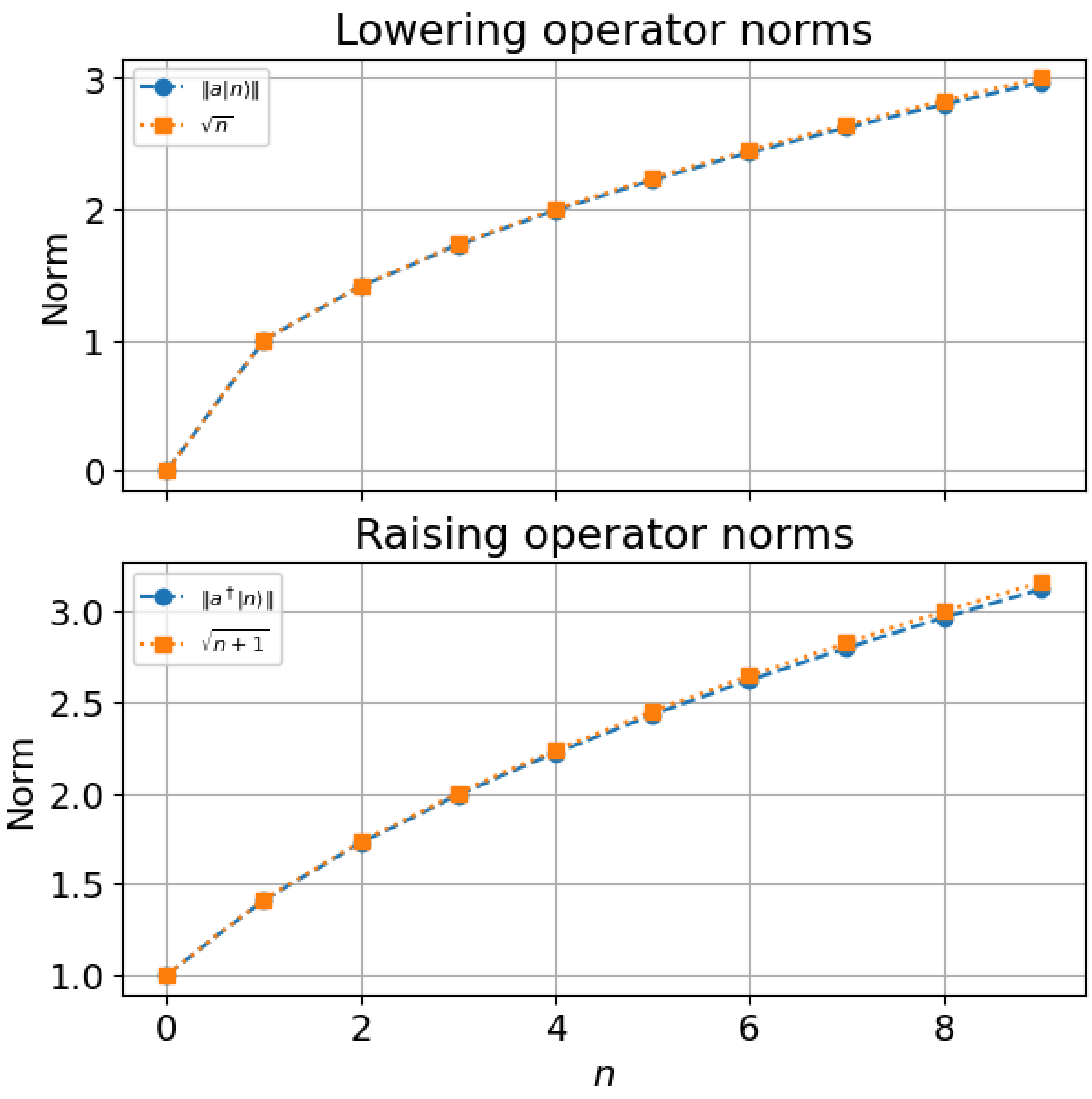
- plotting the norm of the eigenstates (separately for the raising and lower operator) hitting the first ten eigenstates
- plotting $\sqrt{N}$

and checking that they are the same

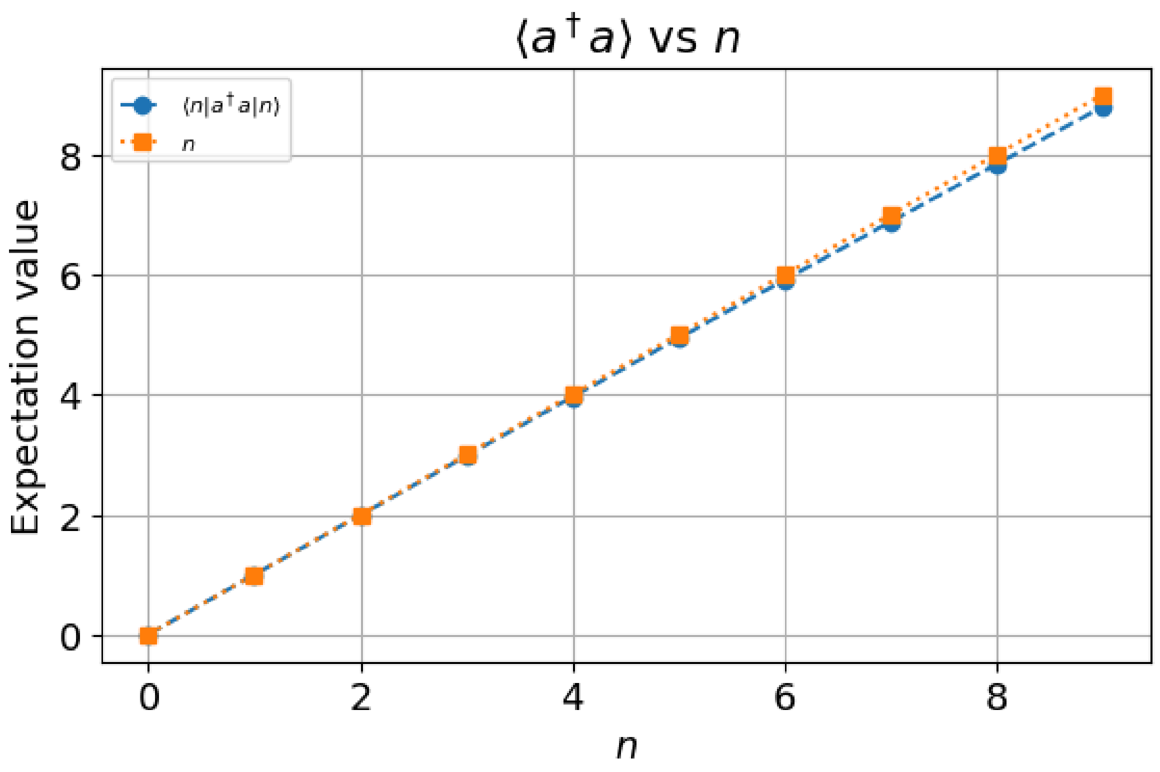
We can also check that $a^\dagger a$ is a number operator. Plot again for the first ten eigenstates, the expectation value of $a^\dagger a$ in those eigenstates.

? Answer (start)

In [ ]:
```
#Getting the raised and lowered norms

n_max = 10
ns = np.arange(n_max)

lower_norms = []
raise_norms = []

for n in ns:
    psi_n = get_state(n)

    lowered = a @ psi_n
    raised  = adag @ psi_n

    lower_norms.append(np.linalg.norm(lowered))
    raise_norms.append(np.linalg.norm(raised))

lower_norms = np.array(lower_norms)
raise_norms = np.array(raise_norms)

#Plotting the numerical values to the theoretical values

fig, axes = plt.subplots(2, 1, figsize=(6, 6), constrained_layout=True, sharex=T
ax1, ax2 = axes

#Lowering norms vs sqrt(n)
ax1.plot(ns, lower_norms, 'o--', label=r'$\|a|n\rangle\|$')
ax1.plot(ns, np.sqrt(ns), 's:', label=r'$\sqrt{n}$')
ax1.set_ylabel('Norm')
ax1.set_title('Lowering operator norms')
ax1.grid(True)
ax1.legend(loc='best', fontsize=8)

#Raising norms vs sqrt(n+1)
ax2.plot(ns, raise_norms, 'o--', label=r'$\|a^\dagger|n\rangle\|$')
ax2.plot(ns, np.sqrt(ns+1), 's:', label=r'$\sqrt{n+1}$')
ax2.set_xlabel(r'$n$')
ax2.set_ylabel('Norm')
ax2.set_title('Raising operator norms')
ax2.grid(True)
ax2.legend(loc='best', fontsize=8)

plt.show()
```



```
In [ ]: # check that a†a is the number operator
        N_op = adag @ a

        #Computing ⟨a†a⟩ for the first ten eigenstates
        N_expect = np.zeros(n_max)

        for n in ns:
            psi_n = get_state(n)
            psi_dag = psi_n.conjugate().T
            N_expect[n] = np.real((psi_dag @ N_op @ psi_n).item())

        #Plotting the expectation values of the number operator
        fig, ax = plt.subplots(1, 1, figsize=(6, 4), constrained_layout=True)

        ax.plot(ns, N_expect, 'o--', label=r'$\langle n|a^\dagger a|n\rangle$')
        ax.plot(ns, ns, 's:', label=r'$n$')

        ax.set_xlabel(r'$n$')
        ax.set_ylabel(r'Expectation value')
        ax.set_title(r'$\langle a^\dagger a \rangle$ vs $n$')
        ax.grid(True)
        ax.legend(loc='best', fontsize=8)

        plt.show()
```



This shows that $\langle a^{\text{\dag}} a\rangle$ is essentially the number operator.

✓ Answer (end)

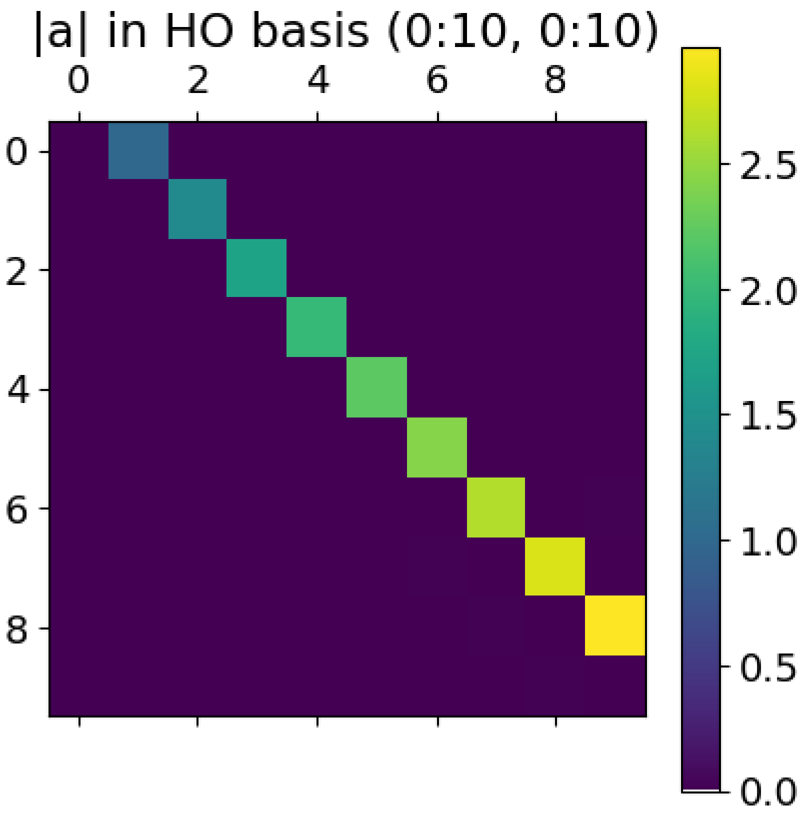
d. Write it in the basis of number operators

Now we would like to go ahead and write $a^\dagger$ and a in the Harmonic Oscillator basis. In this basis, what we expect to see is that both the raising and lowering operators are just above and below the diagonal respectively. To rotate an operator from the position basis into the Harmonic oscillator basis, you want to do `v.T.conjugate() @ O @ v` for an operator `O` where `v` is the eigenstates of the simple harmonic oscillator. Go ahead and rotate a, $a^\dagger$ and $a^\dagger a$ into the Harmonic oscillator basis and look at the top 10×10 chunk. Does it make sense? What are the values of the diagonal of $a^\dagger a$

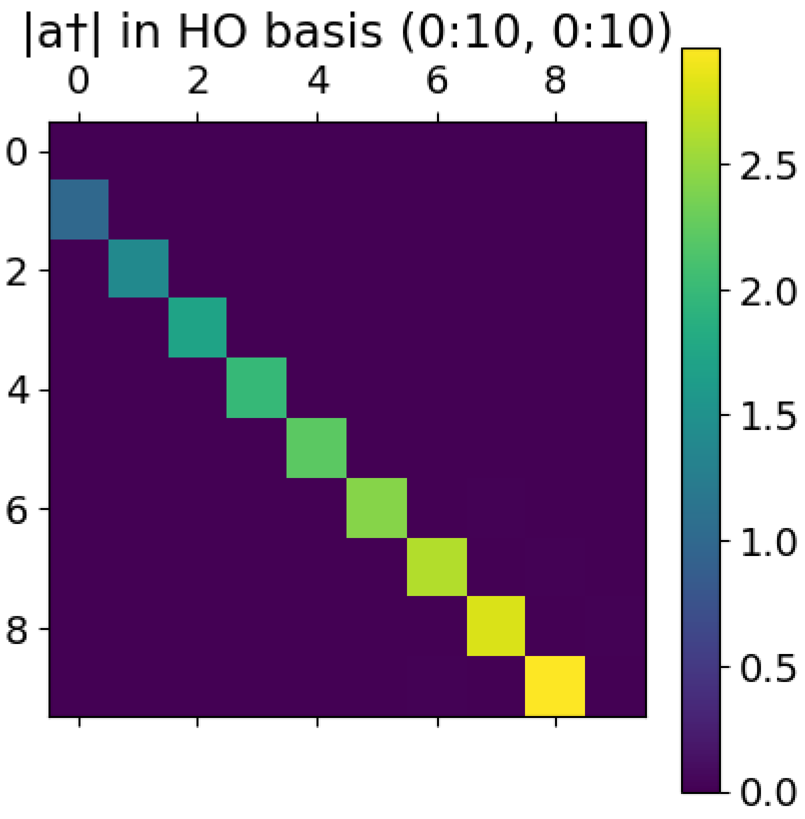
? Answer (start)

```
In [ ]: # Transform to HO eigenbasis (|n> basis)
        a_HO    = v.conjugate().T @ a    @ v   # a in HO basis
        adag_HO = v.conjugate().T @ adag @ v   # a† in HO basis
        N_HO    = v.conjugate().T @ N_op @ v   # a† a in HO basis
```

```
In [ ]: #Top 10x10 of the lowering operator
        plt.matshow(np.abs(a_HO[:10, :10]))
        plt.colorbar()
        plt.title("|a| in HO basis (0:10, 0:10)")
        plt.show()
```



In [ ]:
```
plt.matshow(np.abs(adag_HO[:10, :10]))
plt.colorbar()
plt.title("|a†| in HO basis (0:10, 0:10)")
plt.show()
```



The structure of the Raising and Lowering Operators in the Harmonic Oscillator Basis makes sense because

The harmonic oscillator eigenstates satisfy:

- Lowering operator

$$a|n\rangle = \sqrt{n}\,|n-1\rangle$$

- Raising operator

$$a^\dagger|n\rangle = \sqrt{n+1}\,|n+1\rangle$$

Using the matrix element definition

$$O_{mn} = \langle m|\,O\,|n\rangle,$$

row index = output state, column index = input state.

Hence...

1. Lowering operator (a) is **superdiagonal**

From

$$a|n\rangle = \sqrt{n}\,|n-1\rangle,$$

we get:

- Input: column (n)
- Output: row (n-1)

Thus:

$$a_{n-1,\,n} = \sqrt{n}$$

Since

$$\text{row index} = \text{column index} - 1,$$

the nonzero elements lie just above the main diagonal.

2. Raising operator (a^\dagger) is **subdiagonal**

From

$$a^\dagger|n\rangle = \sqrt{n+1}\,|n+1\rangle,$$

we get:

- Input: column (n)
- Output: row (n+1)

Thus:

$$(a^\dagger)_{n+1,\,n} = \sqrt{n+1}$$

Since

$$\text{row index} = \text{column index} + 1,$$

the nonzero elements lie just below the main diagonal.

In summary

- **Lowering operator (a)** → superdiagonal
- **Raising operator (a^\dagger)** → subdiagonal

This matches the expected ladder structure in the harmonic oscillator energy basis.

In [ ]:
```
#Diagonal of number operator
diag_N = np.diag(N_HO)[:10]
print("Diagonal of a† a in HO basis (first 10):")
print(diag_N)
```

```
Diagonal of a† a in HO basis (first 10):
[9.76318033e-07+0.j 9.97504095e-01+0.j 1.99001070e+00+0.j
 2.97752308e+00+0.j 3.96004349e+00+0.j 4.93757417e+00+0.j
 5.91011730e+00+0.j 6.87767507e+00+0.j 7.84024960e+00+0.j
 8.79784299e+00+0.j]
```

This values are equal to [0, 1, 2, 3, ..., 9] up tu numerical noise. This makes sense because the number operator must be:

$$N = \begin{pmatrix} 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

In [ ]:
```
diag_N_rounded = np.round(diag_N).astype(int)

print("Rounded diagonal of a†a in HO basis (first 10):")
print(diag_N_rounded)
```

```
Rounded diagonal of a†a in HO basis (first 10):
[0 1 2 3 4 5 6 7 8 9]
C:\Users\ariel\AppData\Local\Temp\ipykernel_31832\2717544083.py:1: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  diag_N_rounded = np.round(diag_N).astype(int)
```

✓ Answer (end)

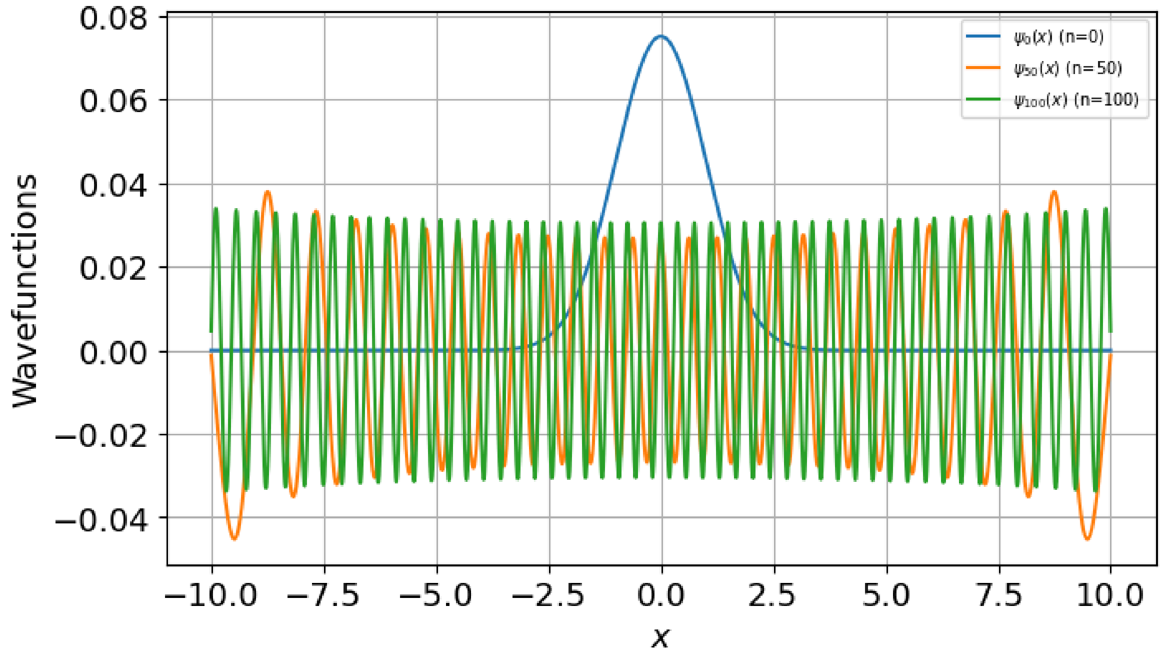
e. Warts

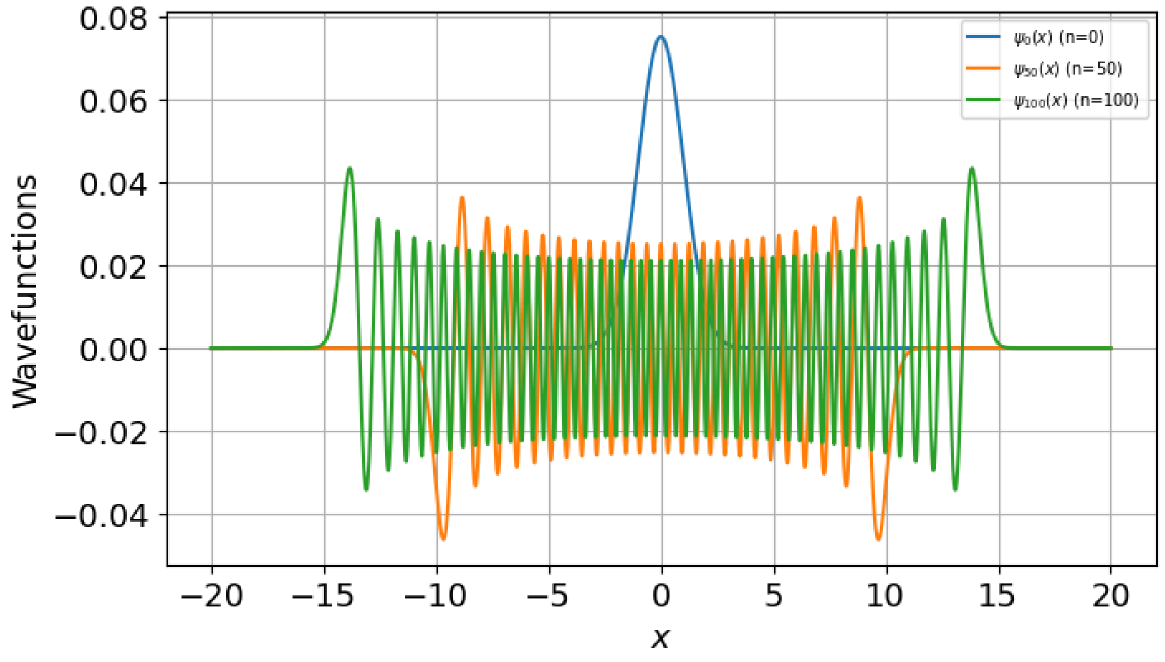
So far everything has worked out well when we discretized things. In this section, we just mention a couple warts that you need to look out for.

- The simplest thing is just that the eigenstates and eigenvalues start diverging from the true answer due to the finite L and discretization. This is very standard but you can see this here by plotting the eigenstates out to $n = 100$. What do you notice? To fix this, you can go ahead and increase $L = 40$ and see that the eigenstates out to $n = 100$ look fine.

? Answer (start)

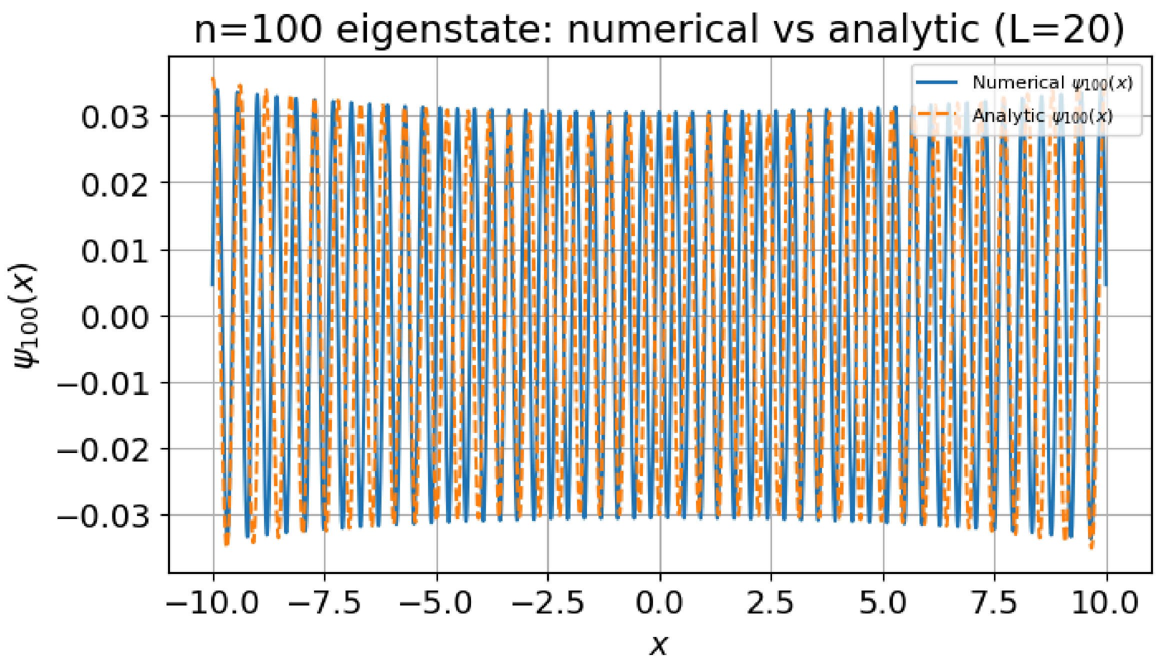
```
In [ ]: delta_x = 0.01

        def build_SHO(L, delta_x):
            xs = SetupGrid(L, delta_x)                         # -L/2 ... L/2
            X, P, P2 = SetupObservables(xs, delta_x)
            V_diag = 0.5 * xs**2                               # V(x) = x^2 / 2
            H = 0.5 * P2 + np.diag(V_diag)
            E, v = np.linalg.eigh(H)                           # eigenvalues, eigenvectors
            return xs, H, E, v,

        def plot_eigenstates_with_potential(L, delta_x, ns=[0, 50, 100]):
            xs, H, E, v = build_SHO(L, delta_x)

            fig, ax = plt.subplots(1, 1, figsize=(7, 4), constrained_layout=True)

            # We scale V to about the max amplitude of the plotted eigenstates.
            max_amp = 0.0
            for n in ns:
                psi_n = v[:, n]
                max_amp = max(max_amp, np.max(np.abs(psi_n)))

            # V_scale = max_amp / np.max(V_diag) if np.max(V_diag) > 0 else 1.0
            # V_plot = V_diag * V_scale

            # Plot the selected eigenstates
            for n in ns:
                psi_n = v[:, n]
                ax.plot(xs, psi_n, label=fr'$\psi_{{{n}}}(x)$ (n={n})')

            # # Plot rescaled potential
            # ax.plot(xs, V_plot, 'k--', label="$V(x)=\\frac{1}{2}x^2$ (scaled)")

            ax.set_xlabel('$x$')
            ax.set_ylabel("Wavefunctions")
            #ax.set_title(f'Harmonic oscillator eigenstates up to n={ns[-1]} (L={L})')
            ax.grid(True)
            ax.legend(fontsize=7, loc='upper right')

            plt.show()

        #Run for L = 20
        plot_eigenstates_with_potential(L=20, delta_x=delta_x)

        #Run for L = 40
        plot_eigenstates_with_potential(L=40, delta_x=delta_x)
```

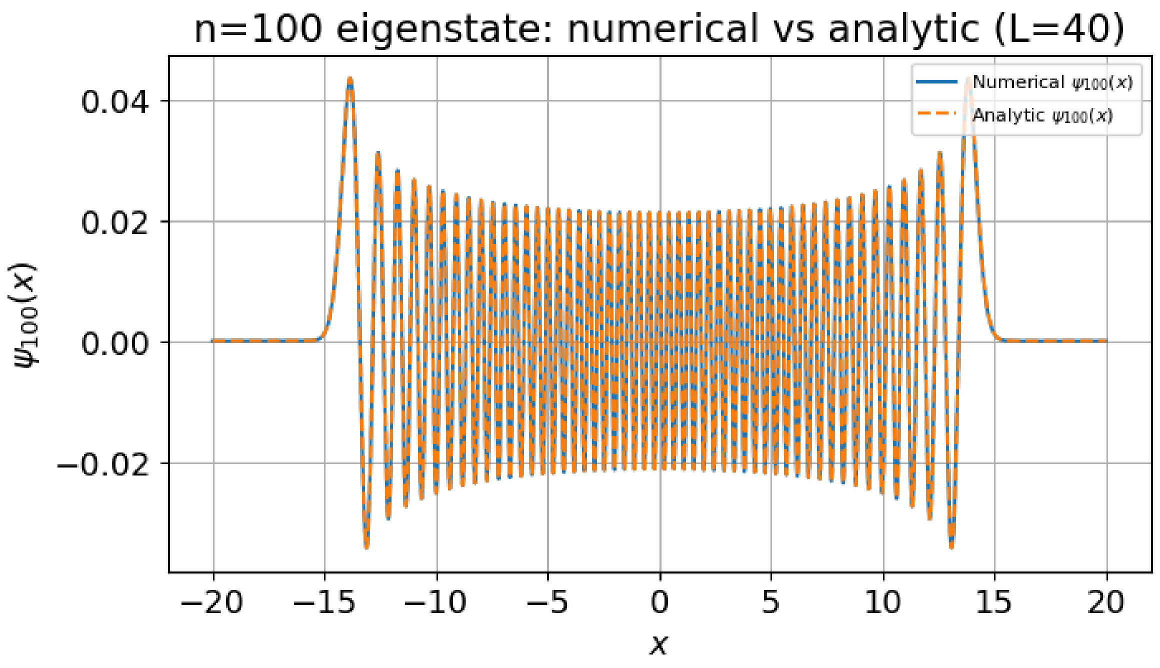




In [ ]:
```python
for L in [20, 40]:
    # Build numerical SHO
    xs, H, E, v = build_SHO(L, delta_x)

    # Numerical n=100 eigenstate
    psi_num = normalize(v[:, n])

    # Analytic n=100 eigenstate on the same grid
    psi_th = HO_analytic_statState(xs, n)

    # Align global phase of analytic state to numerical one
    psi_th_aligned = align_phase(psi_num, psi_th)

    # Plot comparison
    fig, ax = plt.subplots(1, 1, figsize=(7, 4), constrained_layout=True)

    ax.plot(xs, psi_num.real,  label=r'Numerical $\psi_{100}(x)$')
    ax.plot(xs, psi_th_aligned.real, '--', label=r'Analytic $\psi_{100}(x)$')
    ax.set_xlabel('$x$')
    ax.set_ylabel(r'$\psi_{100}(x)$')
    ax.set_title(f'n=100 eigenstate: numerical vs analytic (L={L})')
    ax.grid(True)
    ax.legend(fontsize=8, loc='upper right')

    plt.show()
```

```
C:\Users\ariel\AppData\Local\Temp\ipykernel_31832\2838844749.py:9: DeprecationWar
ning: `np.math` is a deprecated alias for the standard library `math` module (Dep
recated Numpy 1.25). Replace usages of `np.math` with `math`
  NormFac = (1.0/np.pi)**0.25 / np.sqrt(2.0**n * np.math.factorial(n))
```





Comparison of Numerical and Analytic (\psi_{100}(x))

- For (L = 20):
  The numerical eigenstate $\psi_{100}(x)$ deviates significantly from the analytic solution near the edges of the domain.
  This happens because the classical turning points for $n = 100$ lie outside the

interval $[-10, 10]$, so the finite box artificially clips and distorts the wavefunction.

- For (L = 40):
  The numerical and analytic $\psi_{100}(x)$ match extremely well across the entire range. With the wider interval $[-20, 20]$, the eigenstate no longer feels the boundaries, and the numerics reproduce the true Hermite–Gaussian shape.

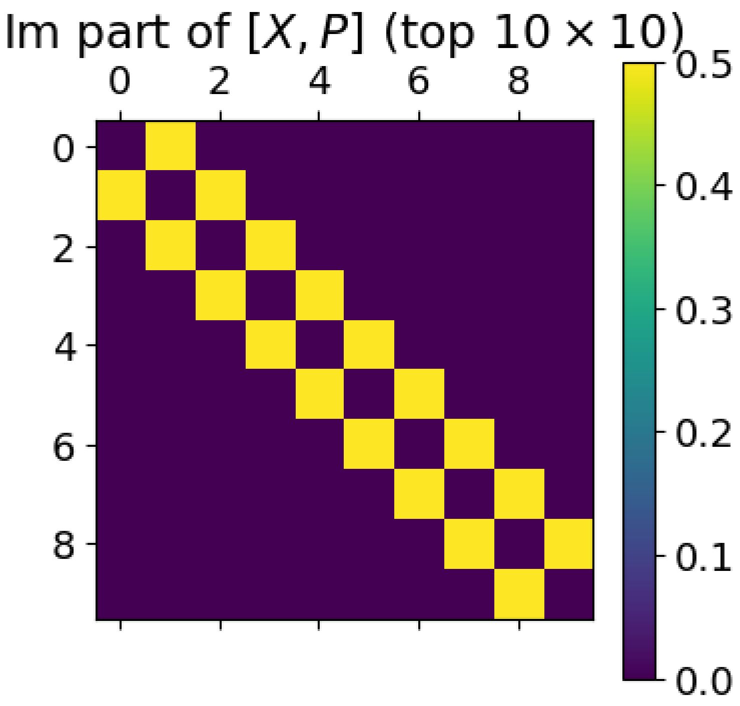
✓ Answer (end)

We learned in quantum mechanics that x and p obey canonical commutation relations - i.e. $[X, P] = i\hbar$ (which means explicitly it's the identity matrix times $i\hbar$).

Let's test this. Take the X and P operators and compute their commutator. Do a matshow of the imaginary part of the top $10 \times 10$ piece of the matrix.

? Answer (start)

In [ ]:
```python
# Compute commutator [X, P]
comm = X @ P - P @ X

# Extract the top-left 10×10 imaginary part
comm_imag = np.imag(comm[:10, :10])

plt.figure(figsize=(4,4))
plt.matshow(comm_imag, cmap='viridis', fignum=False)
plt.colorbar()
plt.title(r'Im part of $[X,P]$ (top $10\times 10$)')
plt.show()
```



✓ Answer (end)

You may notice that you get something that is not diagonal at all. Instead their is $i\hbar$ right off the diagonal. What's going on!?

Remember that the rows and columns are labelled by x. This means that what we expected was

$$\langle x|[\hat{X},\hat{P}]|x\rangle = i\hbar$$

But what we got was the piece that was just off the diagonal having $i\hbar$ on it - i.e. you found that

$$\langle x|[\hat{X},\hat{P}]|x+\Delta x\rangle = \langle x+\Delta x|[\hat{X},\hat{P}]|x\rangle = i\hbar$$

This is very close especially since you could take Δx to be very very tiny. So we almost got the right thing.

What went wrong?

When you actually have finite matrices the trace of a commutator always has to be zero:

$$Tr(XP-PX) = Tr(XP) - Tr(PX) = Tr(XP) - Tr(XP) = 0$$

where the matrices in the trace got flipped because the trace is invariant with respect to cycling matrices. So it's just impossible to have $i\hbar$ down the diagonal. That said, given the closeness it doesn't cause us much trouble.

- Finally, we have learned that

$$H = \frac{1}{2} + \hbar\omega a a^\dagger$$

is the Harmonic Oscillator Hamiltonian.

Go ahead and diagonalize this in the x-basis and plot the first 100 eigenvalues.

You'll notice a set of eigenvalues that you are used to (which can correspond to the eigenvectors you are used to) and an erroneous set of eigenvalues. This is essentially coming from the fact that `P@P` and `P2` are different matrices. You might think that this is some technical issue but it's actually deep and fundamental and causes all sorts of trouble in lattice gauge theory. It goes by the name Fermion doubling and you can see a description of the problem (and the piece that's very relevant here) is you look at this Wikipedia article and look at the section on derivative discretization.

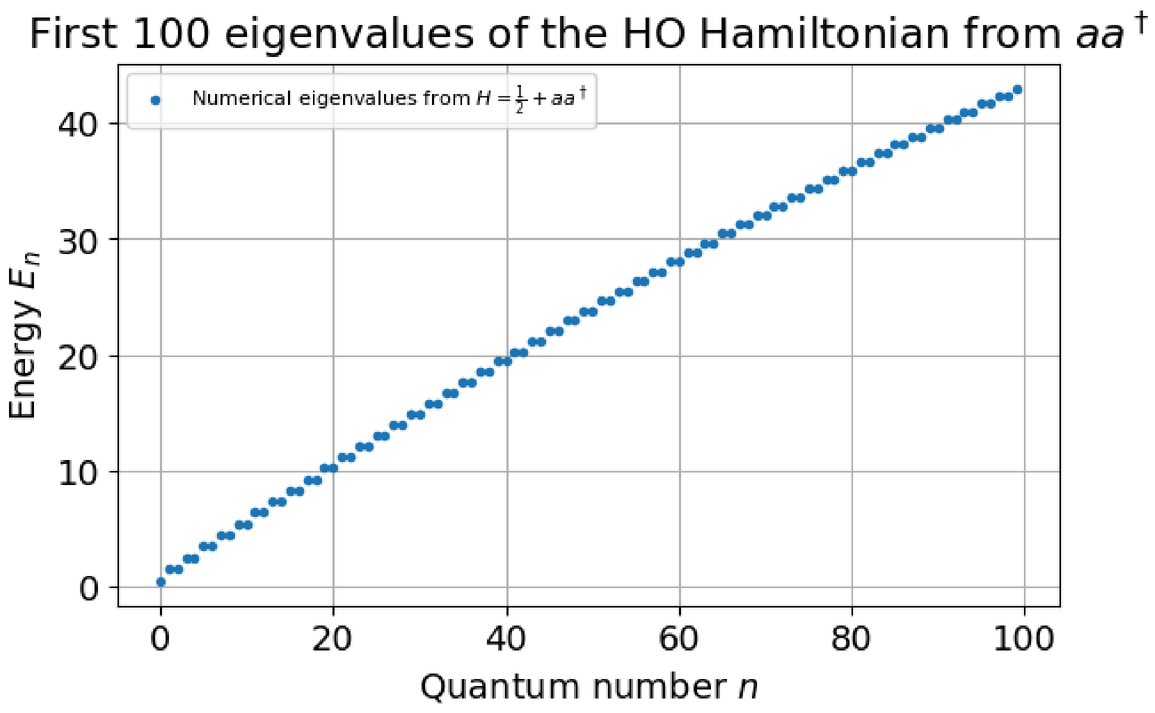
? Answer (start)

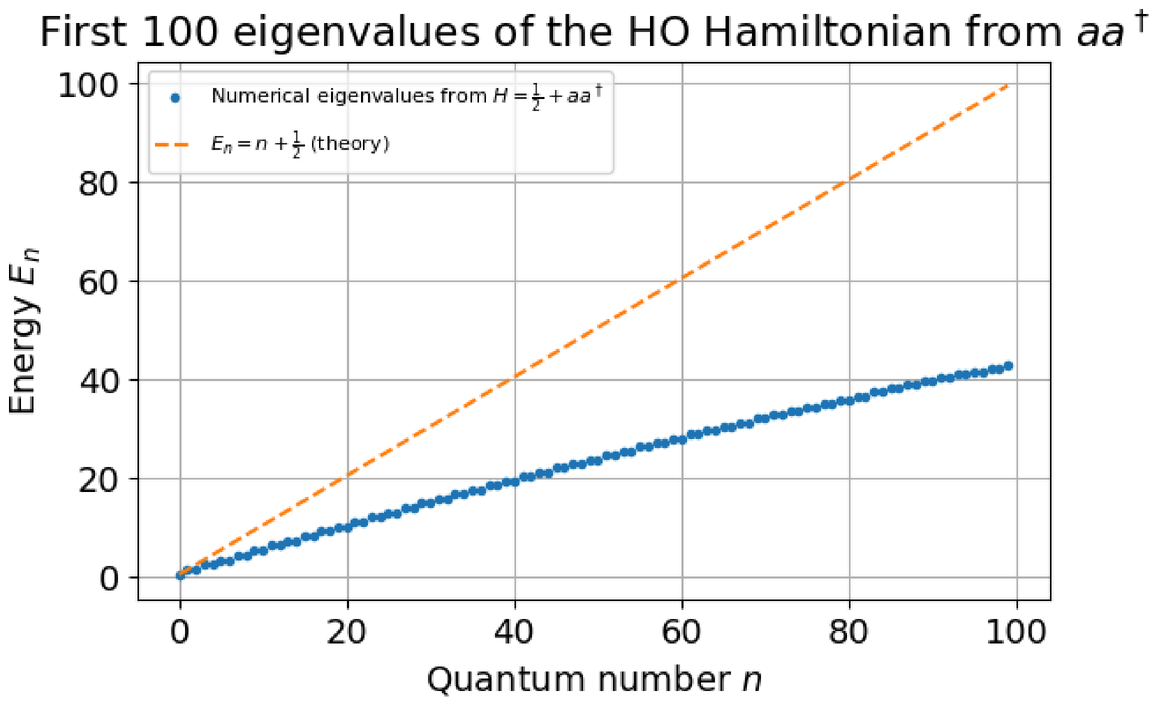
In [ ]:
```
#For this part we extend the domain of x from the usual 20 to 40
delta_x=0.1
L=40
xs=SetupGrid(L,delta_x)
X,P,P2=SetupObservables(xs,delta_x)

# Ladder operators in x-basis (m = ω = ħ = 1)
```

```
a = (X + 1j * P) / np.sqrt(2)
adag = (X - 1j * P) / np.sqrt(2)

N = X.shape[0]

# Hamiltonian built from a a†
H_aa = 0.5 * np.eye(N) + a @ adag

# Diagonalize
E_aa, v_aa = np.linalg.eigh(H_aa)

# Take first 100 eigenvalues
n_max = 100
ns = np.arange(n_max)
E100 = E_aa[:n_max]
```

In [ ]:

```
E_theory = ns + 0.5

# Plot without theoretical comparison
fig, ax = plt.subplots(1, 1, figsize=(6, 4), constrained_layout=True)

ax.plot(ns, E100, '.', label="Numerical eigenvalues from $H = \\frac{1}{2} + aa^

ax.set_xlabel('Quantum number $n$')
ax.set_ylabel('Energy $E_n$')
ax.set_title('First 100 eigenvalues of the HO Hamiltonian from $aa^\dagger$')
ax.grid(True)
ax.legend(fontsize=8, loc='upper left')

plt.show()

# Plot with theoretical comparison
fig, ax = plt.subplots(1, 1, figsize=(6, 4), constrained_layout=True)

ax.plot(ns, E100, '.', label="Numerical eigenvalues from $H = \\frac{1}{2} + aa^
ax.plot(ns, E_theory, '--', label="$E_n = n + \\frac{1}{2}$ (theory)")

ax.set_xlabel('Quantum number $n$')
ax.set_ylabel('Energy $E_n$')
ax.set_title('First 100 eigenvalues of the HO Hamiltonian from $aa^\dagger$')
ax.grid(True)
ax.legend(fontsize=8, loc='upper left')

plt.show()
```


First 100 eigenvalues of the HO Hamiltonian from $aa^\dagger$
Numerical eigenvalues from $H = \frac{1}{2} + aa^\dagger$
Energy E_n
Quantum number n
0
10
20
30
40
0
20
40
60
80
100


First 100 eigenvalues of the HO Hamiltonian from $aa^\dagger$
Numerical eigenvalues from $H = \frac{1}{2} + aa^\dagger$
$E_n = n + \frac{1}{2}$ (theory)
Energy E_n
Quantum number n
0
20
40
60
80
100
0
20
40
60
80
100

✓ Answer (end)