pytalises

Release 0.2.7

May 24, 2021

Contents:

TALISES (This Ain't a LInear Schrödinger Equation Solver) is an easy-to-use Python implementation of the Split-Step Fourier Method, for numeric calculation of a wave function's time-propagation under the Schrödinger equation.

As an introduction we recommend reading Usage and Examples Even more examples can be found here If you want to learn about the employed algorithm for solving the Schrödinger equation read the notes

CHAPTER 1

Usage and Examples

1.1 Freely expanding 1D Gaussian wave packet

Import the pytalises package

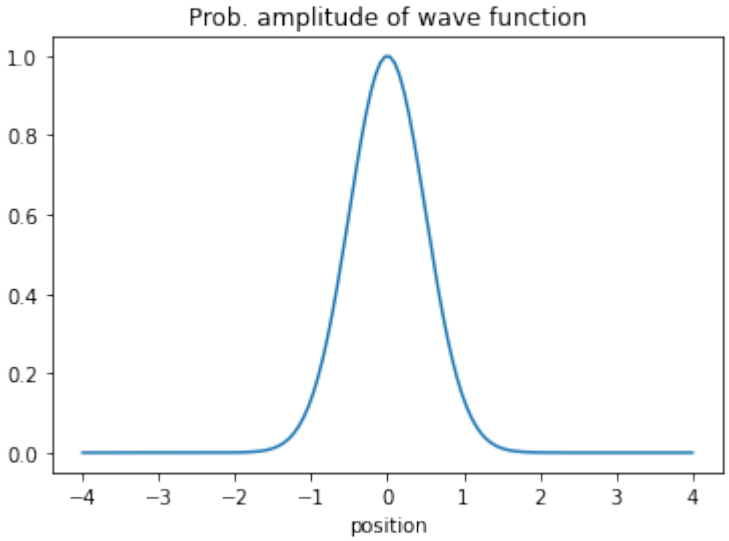
[1]:
```
import pytalises as pt
import numpy as np
import matplotlib.pyplot as plt
```

and instantiate a wave function constituent of 128 complex amplitudes that represent the wave function is position space.

[2]:
```
psi = pt.Wavefunction("exp(-x**2)",
    number_of_grid_points=(128,), spatial_ext=(-4,4))
print(psi.amp.shape)
plt.plot(psi.r, np.abs(psi.amp)**2)
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
```

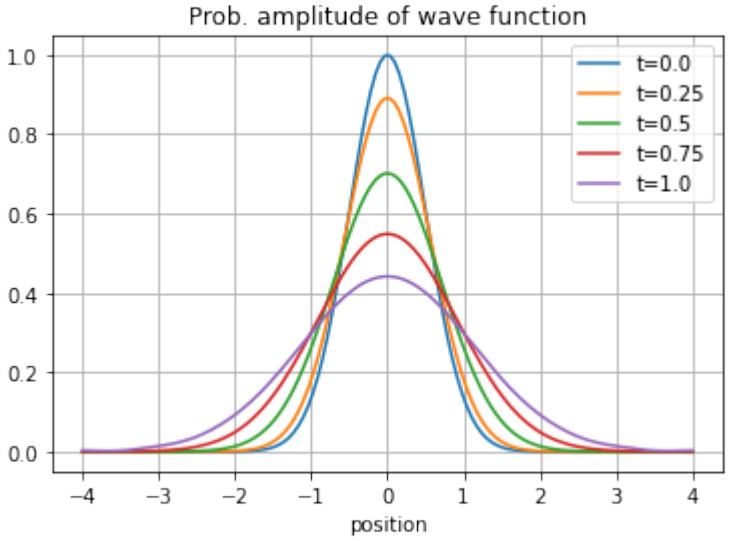
```
(128,)
```

[2]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



The wave packet can be freely propagated (meaning that $V = 0$, or $i\partial_t\psi(r,t) = \frac{\hbar}{2m}\nabla^2\psi(r,t)$) using the `freely_propagate` method.

[3]:
```
for i in range(5):
    plt.plot(psi.r, np.abs(psi.amp)**2, label="t="+str(psi.t))
    psi.freely_propagate(num_time_steps=1, delta_t=0.25)
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
plt.legend()
plt.grid()
```



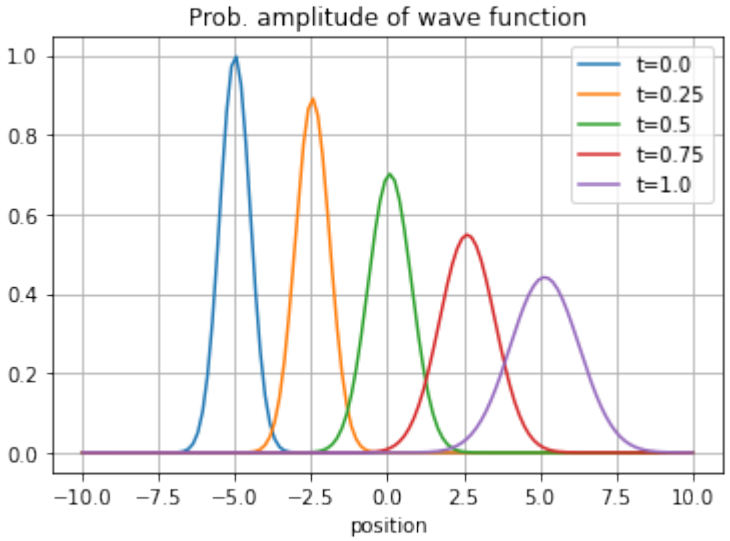
1.2 Free expansion with initial momentum

The wave funciton can be given an initial momentum of k by multiplying it with $\exp(ikx)$.

[4]:
```
psi = pt.Wavefunction('exp(-(x-x0)**2)*exp(1j*k*x)',
    variables={'x0': -5.0, 'k': 10.0}, number_of_grid_points=(128,),
    spatial_ext=(-10,10))
```

The variables that we use in the string to generate the wave function can also be provided by a dictionary as done here. The wave function is offset by $x_0 = -5$.

[5]:
```
for i in range(5):
    plt.plot(psi.r, np.abs(psi.amp)**2, label="t="+str(psi.t))
    psi.freely_propagate(num_time_steps=1, delta_t=0.25)
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
plt.legend()
plt.grid()
```



After one time unit, the wave packet traveled 10 position units from -5 to 5, as expected with momentum $k = 10$.

Attention: This unitless representation is due to the fact that we did not define a mass (optional keyword argument `m`) for the `pytalises.Wavefunction` class. In that case the Schrödinger equation simply became

$$i\partial\psi(r,t) = \frac{1}{2}\nabla^2\psi(r,t).$$

The default value for the `m` keyword argument of the `pytalises.Wavefunction` class is numerically identical to $\hbar$. The Schrödinger equation that is solved in pytalises is actually

$$i\partial\psi(r,t) = \left(\frac{\hbar}{2m}\nabla^2 + \frac{1}{\hbar}V(r,\psi(r,t),t)\right)\psi(r,t).$$

Therefore, always keep in mind that the potential you define $V(r,\psi(r,t),t)$ has to be divided by $\hbar$.

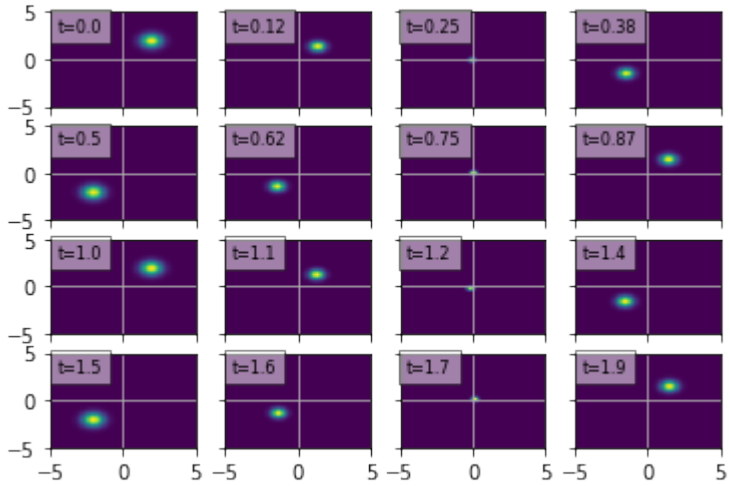
1.3 2D harmonic potential

Now we propagate a wavefunction in a potential $V/\hbar = \frac{1}{2}\omega_x^2 x^2 + \frac{1}{2}\omega_y^2 y^2$ (if not otherwise specified, the mass always equals $\hbar$). Furthermore, we chose $\omega_y = \omega_x = 2\pi 1\mathrm{s}^{-1}$. One period in the harmonic trap takes one second. The Schrödinger equation is then

$$i\partial_t\psi(x,y,t) = \left(\frac{1}{2}\omega^2(x^2+y^2) + \frac{1}{2}\nabla^2\right)\psi(x,y,t).$$

Now we can use the `propagate` method of the `pytalises.Wavefunction` class to do so.

[6]:
```
psi = pt.Wavefunction("exp(-(x-2)**2-(y-2)**2)",
    number_of_grid_points= (128,128), spatial_ext=[(-5,5),(-5,5)])

fig, axs = plt.subplots(4, 4, sharex=True, sharey=True)
for i in range(4):
    for j in range(4):
        axs[i,j].pcolormesh(psi.r[0], psi.r[1], np.abs(psi.amp**2).T,
            rasterized=True)
        axs[i,j].annotate("t={:.2}".format(psi.t), (-4.5,3), fontsize=8)\
            .set_bbox(dict(facecolor='white', alpha=0.5, edgecolor='black'))
        axs[i,j].grid()
        psi.propagate("1/2*omega_x**2*x**2 + 1/2*omega_y**2*y**2",
            variables={'omega_x': 2*np.pi*1, 'omega_y': 2*np.pi*1},
            diag=True, num_time_steps=10, delta_t=0.0125)
```



For better demonstration we also animate the time evolution

[7]:
```
from matplotlib import animation, rc
from IPython.display import HTML
def init():
    im.set_data(np.abs(psi.amp)**2)
    return (im,)

def animate(i):
    psi.propagate("1/2*omega_x**2*x**2 + 1/2*omega_y**2*y**2",
        variables={'omega_x': 2*np.pi*1, 'omega_y': 2*np.pi*1},
        diag=True, num_time_steps=1, delta_t=0.005)
    im.set_data(np.abs(psi.amp)**2)
    return (im,)

psi = pt.Wavefunction("exp(-(x-2)**2-(y-2)**2)",
    number_of_grid_points= (128,128), spatial_ext=[(-5,5),(-5,5)])

fig, ax = plt.subplots()
im = ax.imshow(np.abs(psi.amp)**2,
    vmin=0,
    vmax=10*np.max(np.abs(psi.amp)**2),
    origin='lower')
```

(continues on next page)

(continued from previous page)

```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[7]:
```
<IPython.core.display.HTML object>
```

One can see that the wave packet moves periodically with a frequency of one period per time unit in a diagonal line of the 2D harmonic trap. One could also use the `exp_pos` method of the `Wavefunction` class that returns the mean position of a wave function when called, in order to show the harmonic oscillation.

Note that we called the `propagate` method with the keyword argument `diag=True`. This makes the calculation for the time propagation faster as no numerically diagonalization of the potential energy term is invoked (even though with only one internal state, the potential V can not have any nondiagonal terms).

[8]:
```
def time_propagate(diag):
    psi = pt.Wavefunction("exp(-(x-2)**2-(y-2)**2)",
        number_of_grid_points= (128,128), spatial_ext=[(-5,5),(-5,5)])
    psi.propagate("1/2*omega_x**2*x**2 + 1/2*omega_y**2*y**2",
        variables={'omega_x': 2*np.pi*1, 'omega_y': 2*np.pi*1},
        diag=diag, num_time_steps=10, delta_t=0.0125)
%timeit time_propagate(diag=True)
%timeit time_propagate(diag=False)
```

```
19.1 ms ± 550 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
25.7 ms ± 423 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

1.4 Rabi cycles in two level system

In this example a Gaussian wave packet in a ground state is coherently transferred to an excited state. During this the wave packet will further disperse. The time evolution is described by

$$i\partial_t\psi(x,t) = \left(\frac{1}{2}\begin{pmatrix} 0 & \Omega_R \\ \Omega_R & 0 \end{pmatrix} + \frac{1}{2}\nabla^2\right)\psi(x,t).$$

The well known Rabi model in addition of the kinetic term.

> **Attention**: When defining the list of strings that describe the potential term V, be aware that for the general case you must provide the lower triangular part of V.

For example if

$$V = \begin{pmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix}$$

you must pass the `propagate` method a list `[V11, V21, V22]`. We can omit the other element because $V_{21} = V_{12}^*$ for hermitian matrices. That means if you have a nondiagonal potential V describing the time evolution of a wave function with N_{int} number of internal degrees of freedom, the $N_{\text{int}} \times N_{\text{int}}$ matrix V is described by a list of matrix elements of length $\frac{1}{2}N_{\text{int}}(N_{\text{int}}+1)$.

> **Note**: If you are using `diag=True` in the `propagate` method you only have to provide the diagonal matrix elements V_{ii} of V. Thus, the list is of length N_{int}.

The Rabi frequency will be $\Omega_R = 2\pi f_R = 2\pi 1\text{s}^{-1}$, such that after one time unit a complete population inversion is achieved. We also keep track of the state occupation number of each internal state by calling the `state_occupation` method each timestep.

[9]:
```
def init():
    return (line1,line2,line3,line4,)

def animate(i):
    pop0[i] = psi.state_occupation(0)
    pop1[i] = psi.state_occupation(1)
    psi.propagate(["0", "2*pi*f_R/2", "0"], variables={'f_R': 1, 'pi': np.pi},
        num_time_steps=1, delta_t=delta_t)
    line1.set_ydata(np.abs(psi.amp[:,0])**2)
    line2.set_ydata(np.abs(psi.amp[:,1])**2)
    line3.set_ydata(pop0)
    line4.set_ydata(pop1)
    return (line1,line2,line3,line4,)

psi = pt.Wavefunction(["exp(-x**2)", "0"], number_of_grid_points=(64,),
    spatial_ext=[(-5,5)], normalize_const=1.0 )

fig, axs = plt.subplots(2,1)
line1,line2, = axs[0].plot(psi.r, np.abs(psi.amp[:,0])**2, psi.r, np.abs(psi.amp[:,
↪1])**2)
axs[0].set_xlabel("position")
axs[0].set_ylabel("population density")
axs[0].set_xlim(-5,5)
n_timesteps = 300
delta_t = 0.005
t = np.linspace(0, delta_t*n_timesteps, num=n_timesteps)
pop0 = -np.ones(n_timesteps)
pop1 = -np.ones(n_timesteps)
line3,line4, = axs[1].plot(t, pop0, t, pop1, marker=".", linestyle="", markersize=1)
axs[1].set_xlabel("time")
axs[1].set_ylabel("population")
axs[1].set_ylim(0,1)
axs[1].set_xlim(0,delta_t*n_timesteps)
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=n_timesteps, interval=10, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[9]:
```
<IPython.core.display.HTML object>
```

Note that in this example we have a nondiagonal potential as the eigenstates interact. Thus we can not use the `diag=True` option that was used in the previous example in order to speed up the calculations.

1.5 Diffraction on grating

In this example a 2d Gaussian wave packet is diffracted on a periodic potential. The library that is uses by pytalises to evaluate mathematical expressions that describe wave functions or potentials is numexpr. A list of supported operators and functions that you can use to construct your wave function or potential can be found here. In this specific example we use the `where(cond, a, b)` function for realizing the potential. It either outputs `a` or `b` depending on whether the condition `cond` is fulfilled or not.

Let's plot the wave function and potential

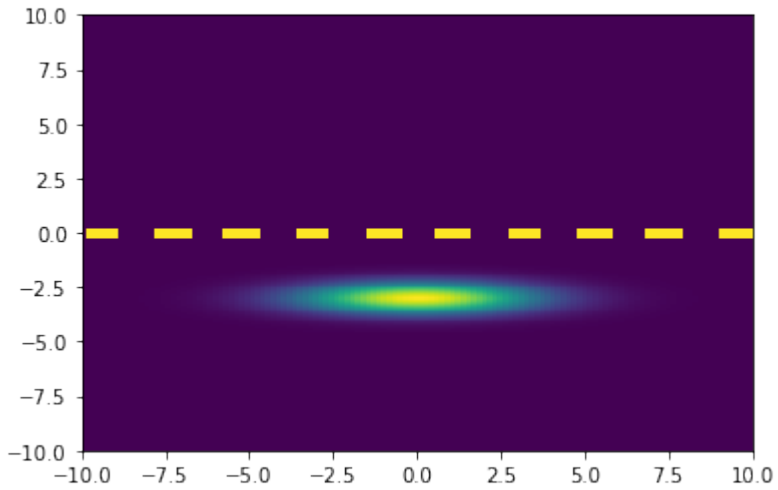
[10]:
```
import numexpr as ne
```

(continues on next page)

(continued from previous page)

```
psi = pt.Wavefunction(
    "exp(-((x-x0)/sigmax)**2)*exp(-((y-y0)/sigmay)**2)*exp(1j*ky*y)",
    variables={'x0': 0, 'y0': -3, 'sigmax':5, 'sigmay': 1, 'ky': 3 },
    number_of_grid_points=(128,256),
    spatial_ext=[(-10,10),(-10,10)],
    )
# String that describes the potential (see numexpr documentaiton for allowed␣
↪functions)
v = "where(y<.2, 1, 0)*where(y>-.2, 1, 0)*where(cos(3*x)<0, 1, 0)*1000"
potential = ne.evaluate(v, local_dict=psi.default_var_dict)[:,:,0]
plt.pcolormesh(psi.r[0], psi.r[1],
    (np.abs(psi.amp**2)+potential).T,
    rasterized=True, vmax=np.max(np.abs(psi.amp**2)))
```

[10]:
```
<matplotlib.collections.QuadMesh at 0x7f355c081cd0>
```



We have given the wave function an initial momentum of $k_y = 3$ and offset it by $y_0 = -3$. After one time unit the center of mass will have collided with the periodic grating.

[11]:
```
def init():
    im.set_data(np.abs(psi.amp.T)**2)
    return (im,)

def animate(i):
    psi.propagate(v, num_time_steps=5, delta_t=0.002, diag=True)
    im.set_data((np.abs(psi.amp**2)+potential).T)
    return (im,)

fig, ax = plt.subplots()
im = ax.imshow((np.abs(psi.amp**2)+potential).T,
    origin='lower', vmax=np.max(np.abs(psi.amp**2)),
    aspect='auto')
anim = animation.FuncAnimation(fig, animate, init_func=init,
    frames=300, interval=10, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[11]:
```
<IPython.core.display.HTML object>
```

1.6 Nonlinear interactions between internal states

In addition to the variables `x`,`y`,`z` and `t` you can use wave function amplitudes in your defined potentials to solve a nonlinear Schrödinger equation

$$i\partial_t\psi(r,t) = \left(\frac{\hbar}{2m}\nabla^2 + \frac{1}{\hbar}V(r,t,\psi(r,t))\right)\psi(r,t)$$

Depending on the number of internal states `N` your wave function has, the wave function amplitudes can be used in the potential by calling `psi0`, `psi1` ... `psiN`.

In this example we create a wave function with two internal states. The first one is going to be stationary in position space whilst the other approaches it and scatters.

The equation that we solve is

$$i\partial_t \begin{bmatrix}\psi_0(x,t)\\ \psi_1(x,t)\end{bmatrix} = \left(\frac{1}{2}\begin{bmatrix}0 & 0\\ 0 & g\cdot|\psi_0(x,t)|^2\end{bmatrix} + \frac{1}{2}\nabla^2\right)\begin{bmatrix}\psi_0(x,t)\\ \psi_1(x,t)\end{bmatrix}.$$

[12]:
```
def init():
    return (line1,line2,)

def animate(i):
    psi.propagate(["0","g/2*abs(psi0)**2"],
        variables={'g': 400},
        num_time_steps=1, delta_t=0.01, diag=True)
    line1.set_ydata(np.abs(psi.amp[:,0])**2)
    line2.set_ydata(np.abs(psi.amp[:,1])**2)
    return (line1,line2,)

psi = pt.Wavefunction(
    ["exp(-(x/2/sigmax)**2)","exp(-((x+x0)/2/sigmax)**2)*exp(1j*k*x)"],
    variables={'x0': 20, 'sigmax':2, 'k': 20 },
    number_of_grid_points=(512,),
    spatial_ext=(-30,30),
    )

fig, ax = plt.subplots()
line1,line2, = ax.plot(psi.r, np.abs(psi.amp[:,0])**2, psi.r, np.abs(psi.amp[:,1])**2)
ax.set_xlabel("position")
ax.set_ylabel("population density")

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=250, interval=10, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[12]:
```
<IPython.core.display.HTML object>
```

Now you should have a good overview of pytalises' capabilities. For more interesting applications we recoomend reading *the additional examples page.*

CHAPTER 2

Additional Examples

In the following section more examples for using pytalises will be explored

[34]:
```
import pytalises as pt
import numpy as np
from matplotlib import pyplot as plt
```

2.1 Time-dependent Rabi model

In *Usage and Examples* the time-independent Rabi model was already simulated. The independence from t is achieved by going into a rotating frame. We will explore this possibility later in other examples. For now, we stick to the regular Schrödinger picture of the Hamiltonian.

The system comprises of two internal states that differ in their energy by $\hbar\omega$. An interaction is driven by a time-periodic potential (e.g. electromagnetic wave). The complete potential is

$$V/\hbar = \begin{bmatrix} 0 & \frac{\Omega}{2}\exp(i\omega t) \\ \frac{\Omega}{2}\exp(-i\omega t) & \omega \end{bmatrix},$$

where Ω is the Rabi frequency.

We generate the wave function and define the Rabi frequency:

[35]:
```
psi = pt.Wavefunction(["exp(-x**2)","0"], number_of_grid_points=(256,),
    spatial_ext=(-5,5), normalize_const=1.0)

V = ["0", "Omega_Rabi/2*exp(-1j*omega*t)", "omega"]

f_Rabi = 1
Omega_Rabi = 2*np.pi*f_Rabi
pulse_length = 1/f_Rabi # One complete inversion
num_time_steps = 100
```

(continues on next page)

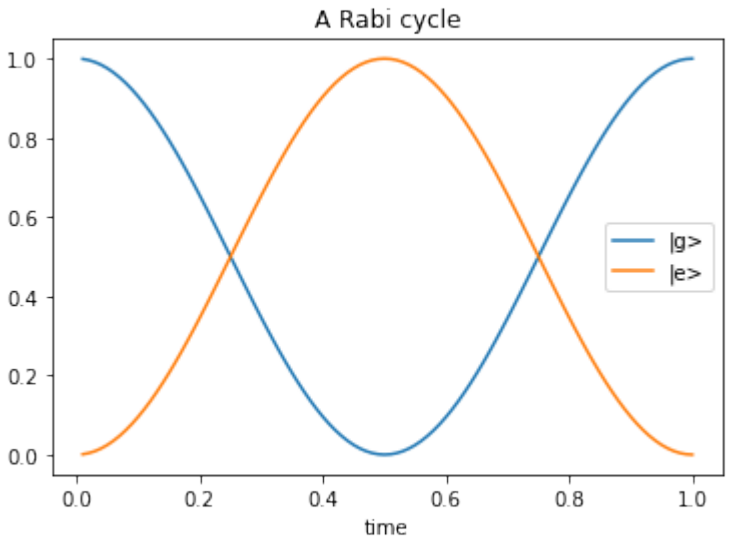
(continued from previous page)

```
pop = np.zeros((num_time_steps, 2)) # vector that saves the state population
time = np.zeros(num_time_steps)
```

We simulate the time-propagation for one time unit. Since the Rabi frequency is 2π we will achieve exactly one inversion.

[36]:
```
for i in range(num_time_steps):
    psi.propagate(V, num_time_steps=1,
        delta_t=pulse_length/num_time_steps,
        variables={'Omega_Rabi': Omega_Rabi, 'omega': 10})
    pop[i,:] = psi.state_occupation()
    time[i] = psi.t

lines = plt.plot(time, pop)
plt.legend(lines, ('|g>', '|e>'))
plt.xlabel("time")
plt.title("A Rabi cycle")
```

[36]:
```
Text(0.5, 1.0, 'A Rabi cycle')
```



2.2 Excitation with momentum transfer

One can also achieve excitation with momentum transfer $|p\rangle \leftrightarrow |p+k\rangle$ with periodic potentials with spatial periodicity k. Ultimately this is what happens with monochromatic laser light that is $\propto \exp i(kx - \omega t)$. Let us look a that in a concrete example:

> **Note**: In many examples we set $\hbar = m$. Thus, the Schrödinger equation svoled is $\partial_t \psi = \frac{1}{2}\nabla^2\psi + \frac{V}{\hbar}\psi$. Furthermore, this implies that in these simulations velocity and wave vector are the same $v = \frac{p}{m} = \frac{\hbar k}{m} = k$. The numeric value for the mass of the simulated particle can be changed with the keyword argument `m` in the `pytalises.Wavefunction` class.

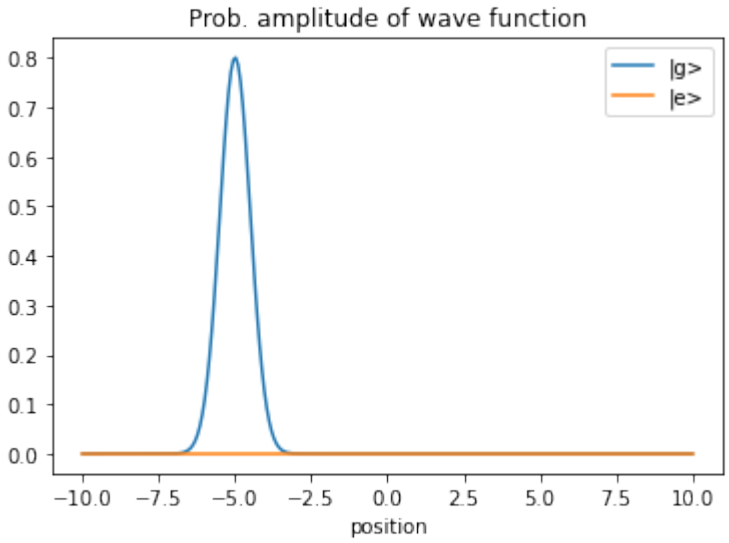
[37]:
```
psi = pt.Wavefunction(["exp(-(x+5)**2)","0"], number_of_grid_points=(256,),
    spatial_ext=(-10,10), normalize_const=1.0)
```

(continues on next page)

(continued from previous page)

```
lines = plt.plot(psi.r, np.abs(psi.amp)**2)
plt.legend(lines, ('|g>', '|e>'))
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
```

[37]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



Again, we have a two level system. We will excite the ground state to the excited state but this time the excited state will gain momentum. The potential for a resonant excitation is

$$V/\hbar = \begin{bmatrix} 0 & \frac{\Omega}{2} \exp i((\omega + \frac{k^2}{2})t + kx) \\ \frac{\Omega}{2} \exp -i((\omega + \frac{k^2}{2})t + kx) & \omega \end{bmatrix}$$

The frequency ω has to be adjusted in order to drive a resonant excitation. There are two reasons for this. Firstly, the momentum transfer results in an increase in kinetic energy $\frac{p^2}{2}$. Furthermore, the state with gained velocity v experiences a Doppler shift of the potential that decreases the seen frequency by $-v \cdot k$. in Our case the initial velocity is zero. The frequency has to be adjusted by $\frac{p^2}{2m\hbar} = \frac{k^2}{2}$.
The pulse will be applied for a quarter the time that it needs for an inversion. In this case we achieve a 50:50 superposition of excited and ground state.

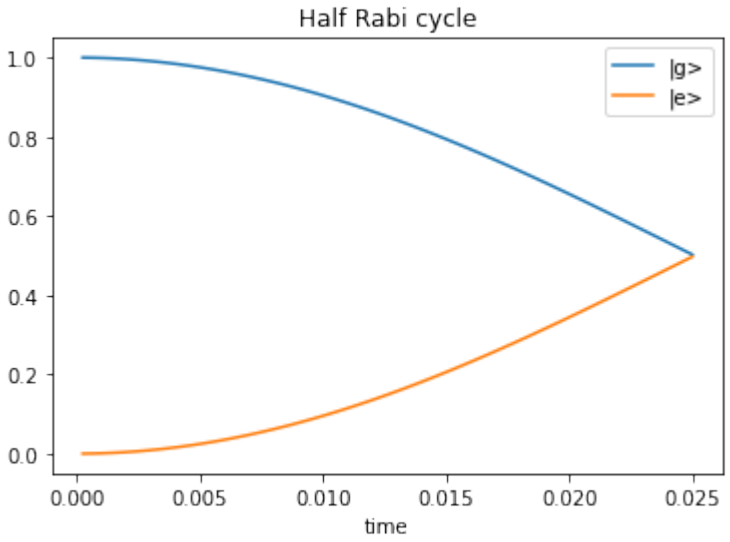
[38]:
```
V = ["0", "Omega_Rabi/2*exp(-1j*((omega+k**2/2)*t-k*x))", "omega"]

f_Rabi = 10
Omega_Rabi = 2*np.pi*f_Rabi
pulse_length = 1/f_Rabi/4 # length for 50:50 beamsplitter pulse
num_time_steps = 100
pop = np.zeros((num_time_steps, 2))
time = np.zeros(num_time_steps)

for i in range(num_time_steps):
    psi.propagate(V, num_time_steps=1,
        delta_t=pulse_length/num_time_steps,
        variables={'Omega_Rabi': Omega_Rabi, 'omega': 10, 'k':10})
    pop[i,:] = psi.state_occupation()
    time[i] = psi.t
```

(continues on next page)

(continued from previous page)

```
lines = plt.plot(time, pop)
plt.legend(lines, ('|g>', '|e>'))
plt.xlabel("time")
plt.title("Half Rabi cycle")
```

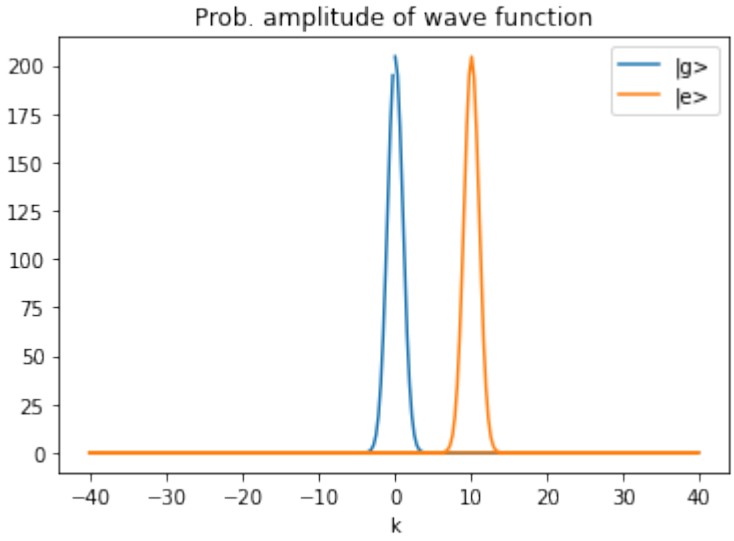
[38]:
```
Text(0.5, 1.0, 'Half Rabi cycle')
```



Indeed, we achieve an equal superposition. Lets have a look at our wave function in momentum space:

[39]:
```
psi.fft()
lines = plt.plot(psi.k, np.abs(psi.amp)**2)
plt.legend(lines, ('|g>', '|e>'))
plt.xlabel("k")
plt.title("Prob. amplitude of wave function")
```

[39]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```

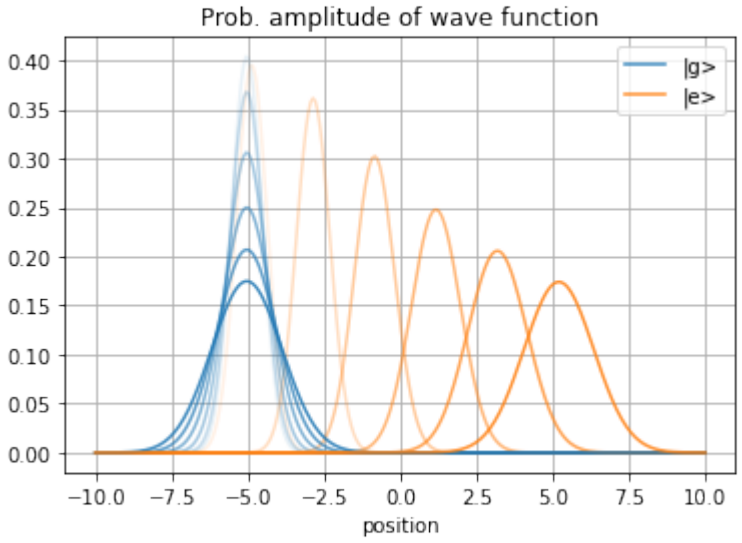


in momentum space we also have an equal superposition of the states $|p\rangle$ and $|p + k\rangle = |p + 10\rangle$. Of course this can

also be seen by looking at the simple free propagation in position space:

[40]:
```
psi.ifft() # transform back into r-space
for i in range(6):
    line1, = plt.plot(psi.r, np.abs(psi.amp[:,0])**2, 'C0-', alpha=.1+i*.15)
    line2, = plt.plot(psi.r, np.abs(psi.amp[:,1])**2, 'C1', alpha=.1+i*.15)
    plt.grid(True)
    psi.freely_propagate(num_time_steps=1, delta_t=0.2)
plt.legend([line1, line2], ('|g>', '|e>'))
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
```

[40]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



2.3 Three-level Raman transitions

In this section we derive the standard three-level Hamiltonian for Raman transitions and simulate the transition. First we do this with no spatial dependency on the electromagnetic field (and therefore no momentum transfer) and then extend this model to the physically relevant situation of imparting a large momentum via two-photon transition on the wave-packet.

Raman transitions with no momentum transfer

The general aim of a Raman transition is to transfer probability amplitudes between two states via a third intermediate state. In this example the three states are $|\omega_g\rangle$, $|\omega_e\rangle$ and $|\omega_i\rangle$. The ground and excited state will be coupled to the intermediate state with monochromatic light of frequencies ω_1 and ω_2, but no direct coupling between the excited and ground state is present.

Δ is the so called one-photon detuning. The Hamiltonian can be written as follow:

[41]:
```
from sympy import *
x, t = symbols('x t', real=True)
Omega_1, Omega_2 = symbols('\Omega_1 \Omega_2', real=True)
k_1, k_2 = symbols('k_1 k_2', real=True)
omega_1, omega_2 = symbols('\omega_1 \omega_2', real=True)
```

(continues on next page)

(continued from previous page)

```
omega_g, omega_e, omega_i = symbols('\omega_g \omega_e \omega_i', real=True)
hbar = symbols('\hbar', constant=True)

H02 = Omega_1/2*exp(I*(k_1*x-omega_1*t))
H12 = Omega_1/2*exp(I*(k_1*x-omega_2*t))
H = Matrix([
        [omega_g,   0,       conjugate(H02)],
        [0,     omega_e,     conjugate(H12)],
        [H02,   H12,         omega_i]])
H
```

[41]:
$$\begin{bmatrix} \omega_g & 0 & \frac{\Omega_1 e^{-i(-\omega_1 t+k_1 x)}}{2} \\ 0 & \omega_e & \frac{\Omega_1 e^{-i(-\omega_2 t+k_1 x)}}{2} \\ \frac{\Omega_1 e^{i(-\omega_1 t+k_1 x)}}{2} & \frac{\Omega_1 e^{i(-\omega_2 t+k_1 x)}}{2} & \omega_i \end{bmatrix}$$

However, there is one problem in using this Hamiltonian for simulations: the frequency differences between the internal states (e.g. $\omega_i - \omega_g$) for optical transitions are at least in the order of 10^{12}Hz. Thus, for simulations with high numerical accuracy, one would need a very short step size in the time domain. In order to relax this we will transform the Hamiltonian into a rotating frame in which the high frequency components will be removed.

The high frequency components of the wave function in the Schrödinger picture are removed by multiplying it with the transformation matrix $\Psi_I = R\Psi$. The Hamiltonian describing the same dynamics is then $H_I = RHR^\dagger - iR\dot{R}^\dagger$. We choose the frequencies of the rotating frame to be

[42]:
```
R = Matrix([
    [exp(I*(omega_i-omega_1)*t),  0,                                  0],
    [0,                  exp(I*(omega_i-omega_2)*t),                  0],
    [0,                  0,                          exp(I*omega_i*t)]
    ])
R
```

[42]:
$$\begin{bmatrix} e^{it(-\omega_1+\omega_i)} & 0 & 0 \\ 0 & e^{it(-\omega_2+\omega_i)} & 0 \\ 0 & 0 & e^{i\omega_i t} \end{bmatrix}$$

R can be somewhat arbitrarily chosen. This choice will yield a Hamiltonian only dependent on Δ. Let us perform the transformation:

[43]:
```
H_I = R*H*conjugate(R) - I*R*conjugate(diff(R,t))
simplify(H_I)
```

[43]:
$$\begin{bmatrix} \omega_1 + \omega_g - \omega_i & 0 & \frac{\Omega_1 e^{-ik_1 x}}{2} \\ 0 & \omega_2 + \omega_e - \omega_i & \frac{\Omega_1 e^{-ik_1 x}}{2} \\ \frac{\Omega_1 e^{ik_1 x}}{2} & \frac{\Omega_1 e^{ik_1 x}}{2} & 0 \end{bmatrix}$$

The time-dependencies on the nondiagonal elements have completely vanished. Looking at the above sketch of the three-level system we see that the diagonal elements are $\omega_1 + \omega_g - \omega_o = -\Delta$, $\omega_2 + \omega_e - \omega_i = -\Delta$ and 0.

This is the potential we will use for the first simulation.

$$V = \begin{bmatrix} -\Delta & 0 & \frac{\Omega_1}{2} \\ 0 & -\Delta & \frac{\Omega_2}{2} \\ \frac{\Omega_1}{2} & \frac{\Omega_2}{2} & 0 \end{bmatrix}$$

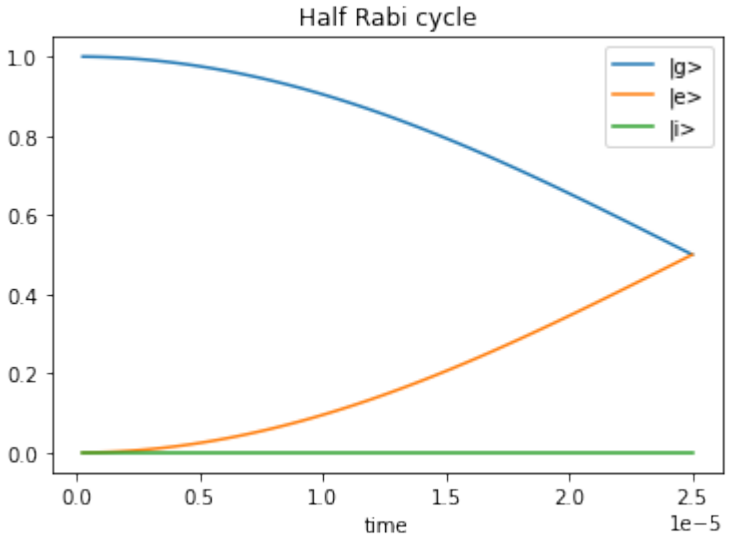
For now we will also neglect the spatial dependencies on k_1 and k_2. We will look at them in the next example when we simulate the transition with momentum transfer.
Furthermore we actually use SI units from now on and simulate these Hamiltonian on a real world example: the Rubidium-87 D_2-line transitions. The intermediate state will be one of the $5^2P_{3/2}$ states. The ground and excited states are the $5^2S_{1/2}$ $F = 1$ and $F = 2$ states. Data for the atom's energy levels can be found here

[44]:
```
m = 1.4447e-25 # Mass of a Rubidium atom

psi = pt.Wavefunction(["exp(-((x-x0)/(2*sigma_x))**2)","0","0"],
    number_of_grid_points=(128,), spatial_ext=(-10e-6,10e-6),
    normalize_const=1.0, m=m,
    variables={'x0': 0, 'sigma_x': 1e-6})

# List of strings describing the lower triangular part of V
V = ["-Delta", "0", "Omega/2", "-Delta", "Omega/2", "0"]

# omega_i-omega_g: energy difference between F'=3 and F=2 Rubidium-87 D2-lines
omega_ig    = 2*np.pi*384.230484468e12
# omega_e-omega_g: energy difference betwwen F=2 and F=1 of the 5^2 S_{1/2} manifold
omega_eg    = 2*np.pi*6.8e9
# One-photon detuning of 700 MHz
Delta       = 2*np.pi*700e6
```

The general Rabi frequency for such two-photon transitions is different from the Rabi frequency of the single-photon transitions. It is $\Omega = \frac{\Omega_1\Omega_2}{2\Delta}$. We calculate the single-photon Rabi frequencies Ω_i from the fact that we aim to achieve Rabi cycles of the two-photon transition of length 100 ms.

[45]:
```
f_Rabi          = 1/100e-6
Omega           = np.sqrt(2*2*np.pi*f_Rabi * Delta)

pulse_length    = 1/f_Rabi/4 # length for beamsplitter
num_time_steps  = 100
pop = np.zeros((num_time_steps, 3))
time = np.zeros(num_time_steps)

for i in range(num_time_steps):
    psi.propagate(V, num_time_steps=1,
        delta_t=pulse_length/num_time_steps,
        variables={'Omega': Omega, 'Delta': Delta})
    pop[i,:] = psi.state_occupation()
    time[i] = psi.t
lines = plt.plot(time, pop)
plt.legend(lines, ('|g>', '|e>', '|i>'))
plt.xlabel("time")
plt.title("Half Rabi cycle")
```

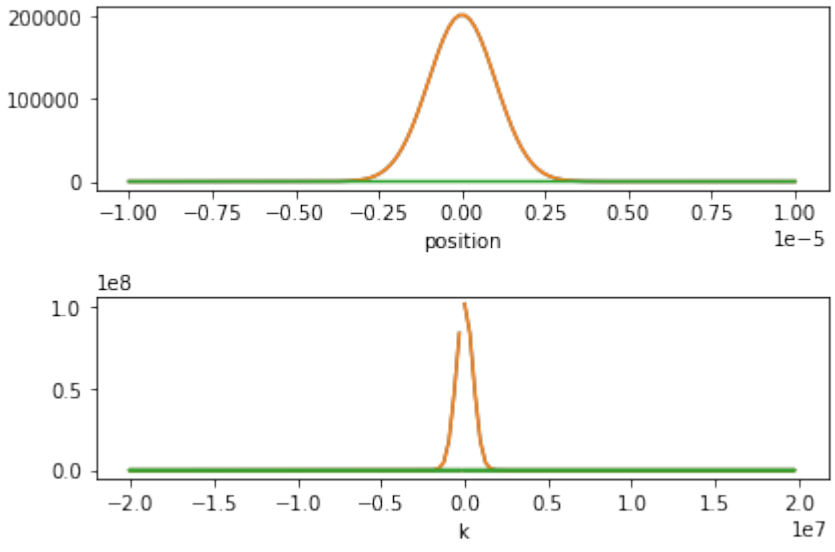
[45]:
```
Text(0.5, 1.0, 'Half Rabi cycle')
```



After 25 μs an equal superposition of ground and excited states is achieved. The intermediate state is only very sparsely populated, but the probability amplitude is indeed transferred via the intermediate state as no nondiagonal elements between ground and excited state exist.

At last let us have a look on the wave function in position and momentum space:

[46]:
```
fig, axs = plt.subplots(2,1)
axs[0].plot(psi.r, np.abs(psi.amp)**2)
axs[0].set_xlabel("position")
psi.fft() # Fouriier transform
axs[1].plot(psi.k, np.abs(psi.amp)**2)
axs[1].set_xlabel("k")
plt.tight_layout()
```



Excited (orange) and ground (blue, but covered by orange) state cover exactly the same position and momentum

states. That is of course boring. In the next calculation we will impart a large momentum on the wave-packet via the two-photon transition.

Raman transitions with momentum transfer

In the previous simulation the wave vectors k_1 and k_2 of the electromagnetic waves were neglected. In another previous example it was also already shown that the off diagonal potential element $\propto \exp ikx$ leads to a momentum transfer (after all $\exp ikx$ is the position/momentum translation operator). Stimulated excitation from $|g\rangle \rightarrow |i\rangle$ and emission $|i\rangle \rightarrow |e\rangle$ with light sources 1 and 2 give the quantum state an additional momentum of $k_1 - k_2$. If k_1 and k_2 are parallel to each other they do not impart a large momentum, but if anti-parallel they do!

This calculation differs also from the previous one for the fact that we have to adjust the energy differences of the lasers as the wave-packet acquires kinetic energy $\frac{p^2}{2m} = \frac{\hbar^2(k_1-k_2)}{2m}$ if undergoing the transition. This additional energy difference between the two light sources will be called δ. The level system is now looking like this:

The resonance condition also includes the term for the Doppler shift $(k_1 - k_2) \cdot v$. Here we assume $v = 0$. The Hamiltonian is the one we derived earlier, but now we include the wave vectors k_i.

We can almost completely proceed as done in the case without momentum transfer, only that we have to solve the equation for the resonance condition to obtain a valid k_2 (the rest will be assumed as given).

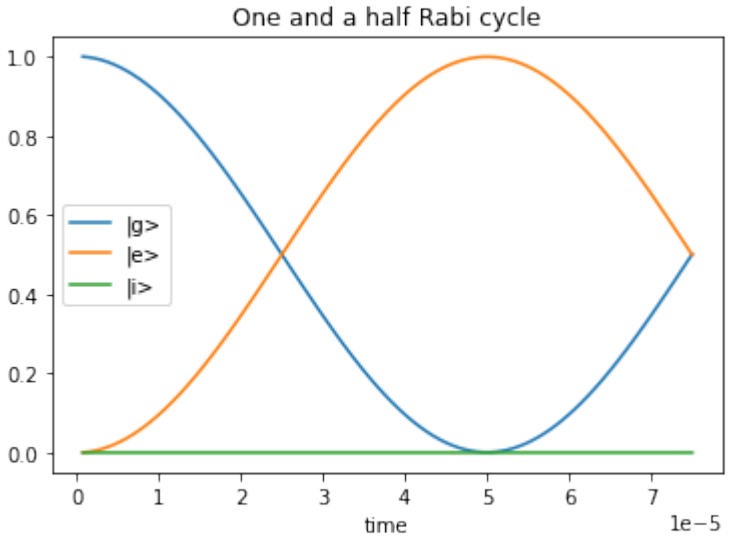
[47]:
```
m = 1.4447e-25
hbar = 1.0545718e-34
c = 299792458

psi = pt.Wavefunction(["exp(-((x-x0)/(2*sigma_x))**2)","0","0"],
    number_of_grid_points=(512,), spatial_ext=(-20e-6,20e-6),
    normalize_const=1.0, m=m,
    variables={'x0': 0, 'sigma_x': 3e-6})

V = ["-Delta", "0", "Omega_Rabi/2*exp(1j*k_1*x)", "-(Delta+delta)", "Omega_Rabi/
↪2*exp(1j*k_2*x)", "0"]

omega_ig    = 2*np.pi*384.230484468e12
omega_eg    = 2*np.pi*6.8e9
Delta       = 2*np.pi*700e6
f_Rabi      = 1/100e-6
Omega_Rabi  = np.sqrt(2*2*np.pi*f_Rabi * Delta)
omega_1     = omega_ig-Delta
k_1         = omega_1/c
# solve resonance condition to acquire k_2
from scipy.optimize import root
k_2         = root(lambda k_2: hbar/2/m*(k_1-k_2)**2 - (omega_1 - c*np.abs(k_2)) +␣
↪omega_eg, -k_1).x
omega_2     = c*np.abs(k_2)
delta    = omega_1-omega_2-omega_eg

pulse_length = 3/f_Rabi/4 # length for 3*pi/2 beamsplitter pulse, just for fun
num_time_steps = 100
pop = np.zeros((num_time_steps, 3))
time = np.zeros(num_time_steps)
```

(continues on next page)

(continued from previous page)

```
for i in range(num_time_steps):
    psi.propagate(V, num_time_steps=1,
        delta_t=pulse_length/num_time_steps,
        variables={'Omega_Rabi': Omega_Rabi, 'c': c,
        'omega_1': omega_1, 'k_1': k_1,
        'omega_2': omega_2, 'k_2': k_2,
        'Delta': Delta, 'delta': delta})
    pop[i,:] = psi.state_occupation()
    time[i] = psi.t
lines = plt.plot(time, pop)
plt.legend(lines, ('|g>', '|e>', '|i>'))
plt.xlabel("time")
plt.title("One and a half Rabi cycle")
```

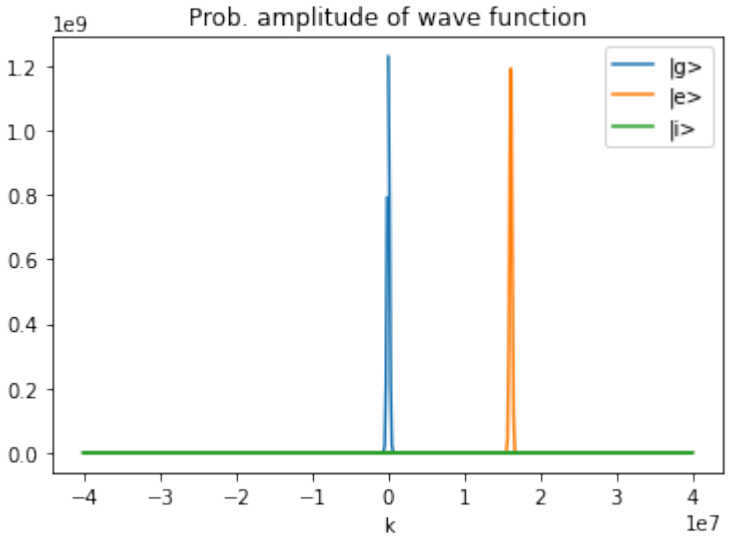
[47]:
```
Text(0.5, 1.0, 'One and a half Rabi cycle')
```



Now if we take a look at the wave function ampltidues in k-space we will see that they are not only in a superposition of energy states, but also momentum states.

[48]:
```
psi.fft()
lines = plt.plot(psi.k, np.abs(psi.amp)**2)
plt.legend(lines, ('|g>', '|e>', '|i>'))
plt.xlabel("k")
plt.title("Prob. amplitude of wave function")
```

[48]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



The momentum is

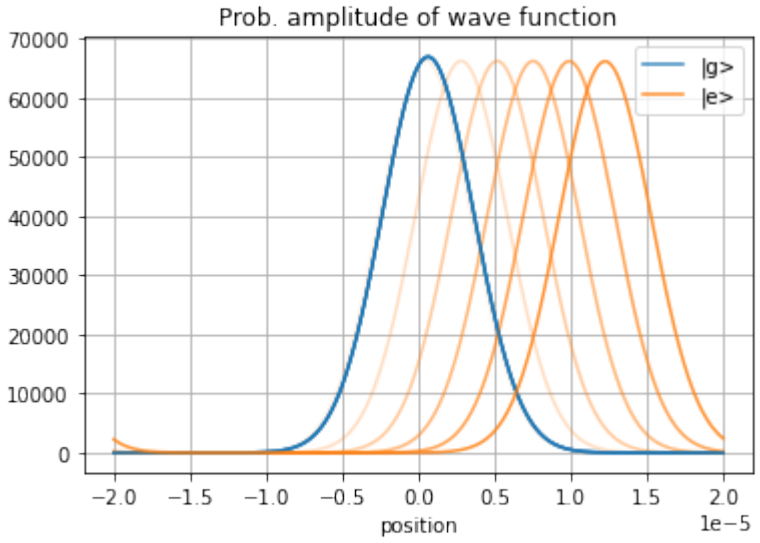
[49]:
```
print(k_1-k_2)
```

```
[16105579.10347923]
```

just as expected.
Let us have a look now how the wave function evolves in position space

[50]:
```
psi.ifft() # transform back into r-space
for i in range(6):
    line1, = plt.plot(psi.r, np.abs(psi.amp[:,0])**2, 'C0-', alpha=.1+i*.15)
    line2, = plt.plot(psi.r, np.abs(psi.amp[:,1])**2, 'C1', alpha=.1+i*.15)
    plt.grid(True)
    psi.freely_propagate(num_time_steps=1, delta_t=2e-4)
plt.legend((line1, line2), ('|g>', '|e>'))
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
```

[50]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



2.4 Single-Bragg diffraction

A very similar case to that of the three-level Raman transition is that of Bragg diffraction. Instead of coupling two different states to achieve a momentum transfer, the Bragg transition just requires a ground state that is coupled to an intermediate state with far detuned lasers.

Again, we will derive the Hamiltonian in the rotating frame. The Hamiltonian is a 2×2 matrix with off diagonal elements constituent of the eletro-magnetic field of the two laser sources.

[51]:
```
x, t = symbols('x t', real=True)
Omega_1, Omega_2 = symbols('\Omega_1 \Omega_2', real=True)
k_1, k_2 = symbols('k_1 k_2', real=True)
omega_1, omega_2 = symbols('\omega_1 \omega_2', real=True)
omega_g, omega_i = symbols('\omega_g \omega_i', real=True)
hbar = symbols('\hbar', constant=True)

H01 = simplify((Omega_1/2*exp(I*(k_1*x-omega_1*t)) + Omega_2/2*exp(I*(k_2*x-omega_
↪2*t))))

H = Matrix([
        [omega_g,       conjugate(H01)  ],
        [H01,           omega_i         ]])
H
```

[51]:
$$\begin{bmatrix} \omega_g & \frac{\Omega_1 e^{i\omega_1 t - ik_1 x}}{2} + \frac{\Omega_2 e^{i\omega_2 t - ik_2 x}}{2} \\ \frac{\Omega_1 e^{-i\omega_1 t + ik_1 x}}{2} + \frac{\Omega_2 e^{-i\omega_2 t + ik_2 x}}{2} & \omega_i \end{bmatrix}$$

Applying the rotating frame yields

[52]:
```
R = Matrix([[exp(I*(omega_i-omega_1)*t), 0],
    [0, exp(I*(omega_i+0*omega_2)*t)]])
H_I = R*H*conjugate(R) - I*R*conjugate(diff(R,t))
simplify(H_I)
```

[52]:
$$\begin{bmatrix} \omega_1 + \omega_g - \omega_i & \frac{\left(\Omega_1 e^{i(\omega_1 t - k_1 x)} + \Omega_2 e^{i(\omega_2 t - k_2 x)}\right) e^{-i\omega_1 t}}{2} \\ \frac{\left(\Omega_1 e^{i(-\omega_1 t + k_1 x)} + \Omega_2 e^{i(-\omega_2 t + k_2 x)}\right) e^{i\omega_1 t}}{2} & 0 \end{bmatrix}$$

Substituting the definitions from the sketch for Δ and δ this gives

$$H_I = \begin{bmatrix} -\Delta & \frac{\Omega_1}{2}\exp(ik_1 x) + \frac{\Omega_2}{2}\exp(i(k_2 x + \delta t)) \\ \frac{\Omega_1}{2}\exp(-ik_1 x) + \frac{\Omega_2}{2}\exp(-i(k_2 x + \delta t)) & 0 \end{bmatrix}$$

[53]:
```
from scipy.optimize import root
m = 1.4447e-25
hbar = 1.0545718e-34
c = 299792458

psi = pt.Wavefunction(["exp(-((x-x0)/(2*sigma_x))**2)","0"],
    number_of_grid_points=(256,), spatial_ext=(-10e-6,10e-6),
    normalize_const=1.0, m=m,
    variables={'x0': 0, 'sigma_x': 1e-6})

V = ["-Delta", "Omega/2 * (exp(1j*k_1*(x+v*t))+exp(1j*(k_2*(x+v*t)+delta*t)))", "0"]

v = 10
omega_ig    = 2*np.pi*384.230484468e12
Delta       = 2*np.pi*700e6
f_Rabi      = 1/300e-6
Omega       = np.sqrt(2*2*np.pi*f_Rabi * Delta)
omega_1     = omega_ig-Delta
k_1         = omega_1/c
k_2         = root(lambda k_2: hbar/2/m*(k_1-k_2)**2 + (k_1-k_2)*v - (omega_1 - c*np.
↪abs(k_2)) , -k_1).x
omega_2     = c*np.abs(k_2)
delta       = omega_1-omega_2
pulse_length = 1/f_Rabi/4 # length for 50:50 beamsplitter pulse
num_time_steps = 100
```

The parameters are the same as in the other simulations and based on realistic experiments on Rubidium-87 atoms driving transitions between the D_2 lines.

Let us finally simulate the time evolution:

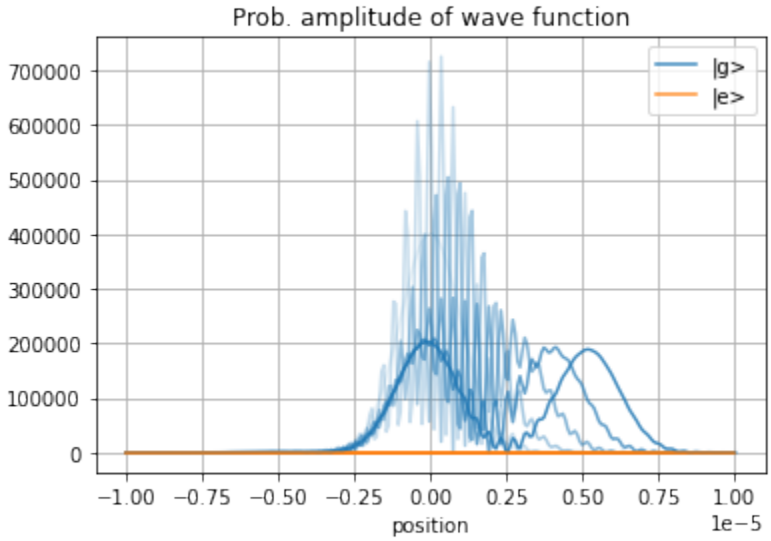
[54]:
```
plt.plot(psi.r, np.abs(psi.amp[:,0])**2, 'C0-', alpha=.1)
plt.plot(psi.r, np.abs(psi.amp[:,1])**2, 'C1', alpha=.1)

psi.propagate(V, num_time_steps=num_time_steps,
    delta_t=pulse_length/num_time_steps,
    variables={'Omega': Omega, 'c': c, 'v': v,
    'omega_1': omega_1, 'k_1': k_1,
    'omega_2': omega_2, 'k_2': k_2,
    'Delta': Delta, 'delta': delta})

for i in range(1,6):
    line1, = plt.plot(psi.r, np.abs(psi.amp[:,0])**2, 'C0-', alpha=.1+i*.15)
```

(continues on next page)

(continued from previous page)

```
    line2, = plt.plot(psi.r, np.abs(psi.amp[:,1])**2, 'C1', alpha=.1+i*.15)
    plt.grid(True)
    psi.freely_propagate(num_time_steps=1, delta_t=1e-4)
plt.legend([line1, line2], ('|g>', '|e>'))
plt.xlabel("position")
plt.title("Prob. amplitude of wave function")
```

[54]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```

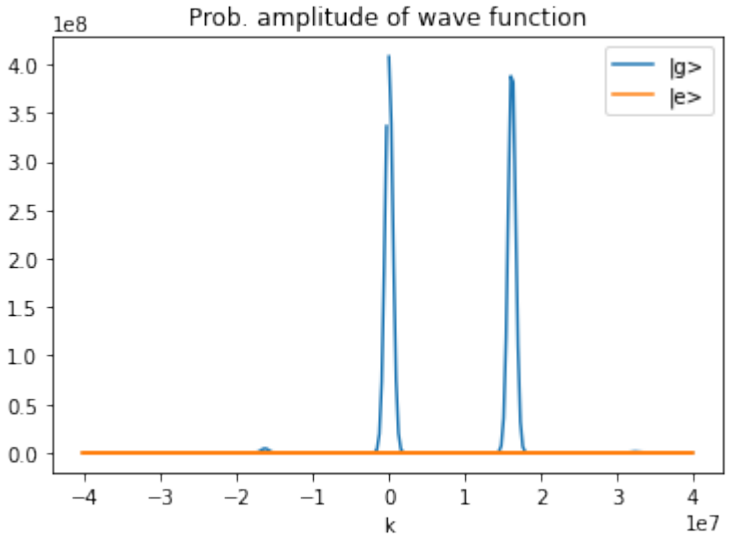


As one can see the excited state is almost not populated, but a superposition of momentum states in the ground state is achieved. As the wave-packets separate they shortly interfere.

Looking at the momentum representation

[55]:
```
psi.fft()
line = plt.plot(psi.k, np.abs(psi.amp)**2)
plt.legend(line, ('|g>', '|e>'))
plt.xlabel("k")
plt.title("Prob. amplitude of wave function")
```

[55]:
```
Text(0.5, 1.0, 'Prob. amplitude of wave function')
```



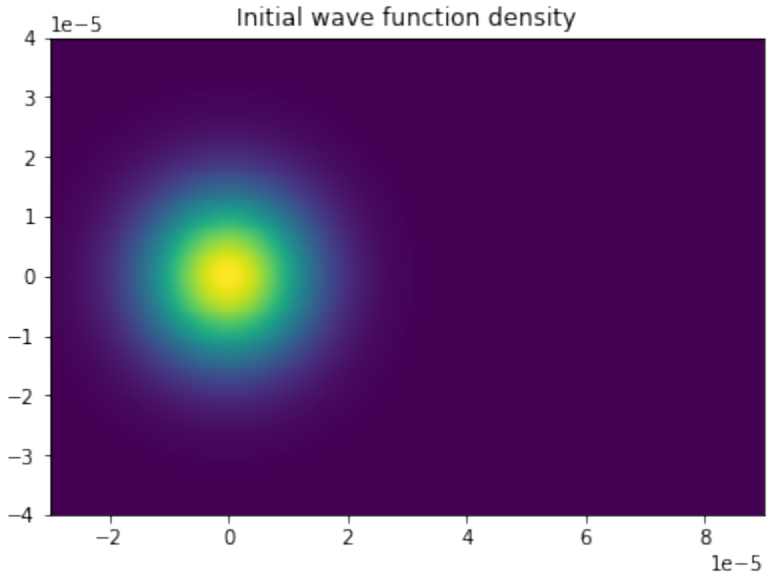
one can see that there is also a very small population in the momentum state $-(k_1 - k_2)$. This is due to the finite pulse length of the laser pulse. Shorter pulses populate different momentum states even more. Try it out!

2.5 2D Single-Bragg diffraction with Gaussian beam

Let's make the previous example slightly more realistic with a 2D Gaussian beam.
The wave function is a 2D Gaussian wave packet

[56]:
```
psi = pt.Wavefunction(["exp(-((x-x0)/(2*sigma_x))**2)*exp(-((y-y0)/(2*sigma_y))**2)",
↪"0"],
    number_of_grid_points=(2048,256), spatial_ext=[(-30e-6,90e-6),(-40e-6,40e-6)],
    normalize_const=1.0, m=m,
    variables={'x0': 0e-6, 'sigma_x': 10e-6, 'y0': 0e-6, 'sigma_y': 10e-6})

fig = plt.figure(figsize=(6.4, 4.8))
ax = fig.add_subplot()
ax.pcolormesh(psi.r[0], psi.r[1], np.abs(psi.amp[:,:,0].T)**2, rasterized=True)
ax.set_aspect("equal")
ax.set_title("Initial wave function density")
```

[56]:
```
Text(0.5, 1.0, 'Initial wave function density')
```



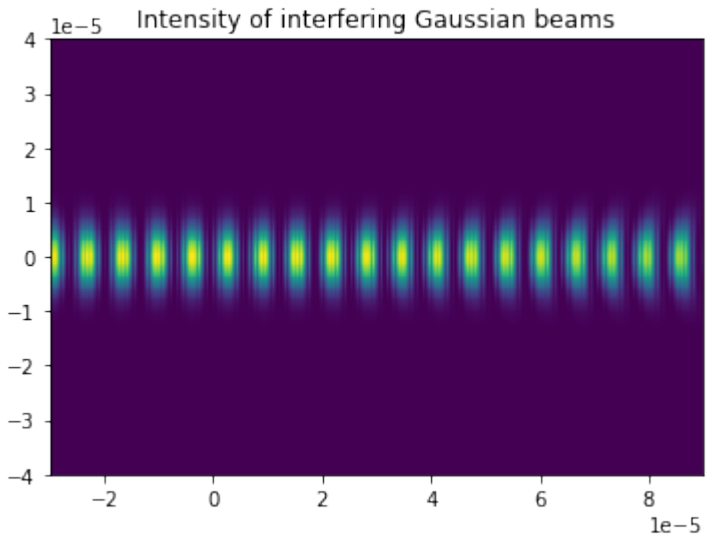
that will be subjected to this more realistic Bragg transition.
The eletric field amplitudes from the prevous example will be exchanged for that of a Gaussian beam. The other parameter will stay the same.

[57]:
```
m = 1.4447e-25
hbar = 1.0545718e-34
c = 299792458

v = 0
omega_ig     = 2*np.pi*384.230484468e12
Delta        = 2*np.pi*700e6
f_Rabi       = 1/400e-6
Omega        = np.sqrt(2*2*np.pi*f_Rabi * Delta)
omega_1      = omega_ig-Delta
k_1          = omega_1/c
k_2          = root(lambda k_2: hbar/2/m*(k_1-k_2)**2 + (k_1-k_2)*v - (omega_1 - c*np.
↪abs(k_2)) , -k_1).x
omega_2      = c*np.abs(k_2)
delta        = omega_1-omega_2
pulse_length = 1/f_Rabi/4 # length for 50:50 beamsplitter pulse
num_time_steps = 10

w0_1         = 7.5e-6 # width of first Gaussian beam
zR_1         = w0_1**2 * k_1 / 2 # Rayleigh range
w0_2         = 7.5e-6 # width of second Gaussian beam
zR_2         = w0_2**2 * k_2 / 2

Amp_gauss_1 = "1/sqrt(1+(x/zR_1)**2) * exp(-(y/(w0_1*sqrt(1+(x/zR_1)**2)))**2 + 1j*(k_
↪1*y**2/(2*x*(1+(zR_1/x)**2)) + k_1*x - arctan(x/zR_1)) )"
Amp_gauss_2 = "1/sqrt(1+(x/zR_2)**2) * exp(-(y/(w0_2*sqrt(1+(x/zR_2)**2)))**2 + 1j*(k_
↪2*y**2/(2*x*(1+(zR_2/x)**2)) + k_2*x - arctan(x/zR_2)) )"

V = ["-Delta", "Omega/2 * ("+Amp_gauss_1+"+"+Amp_gauss_2+"*exp(1j*delta*t) )", "0"]
```

We take a look at the absolute magnitude of the potential's nondiagonal element:

[58]:
```
U = pt.Propagator(psi, V, variables={'Omega': Omega, 'c': c, 'v': v,
    'omega_1': omega_1, 'k_1': k_1, 'w0_1': w0_1, 'zR_1': zR_1,
    'omega_2': omega_2, 'k_2': k_2, 'w0_2': w0_2, 'zR_2': zR_2,
    'Delta': Delta, 'delta': delta})
U.eval_V()

fig = plt.figure()
ax = fig.add_subplot()
ax.pcolormesh(psi.r[0], psi.r[1], np.abs(U.V_eval_array[:,:,0,1,0].T)**2,␣
↪rasterized=True)
ax.set_aspect("equal")
ax.set_title("Intensity of interfering Gaussian beams")
```

[58]:
```
Text(0.5, 1.0, 'Intensity of interfering Gaussian beams')
```



(Note that we used the `pytalises.Propagator` class for this. This class is used in the backend to propagate a `pytalises.Wavefunction` object, but otherwise not used in the examples. You can read more about it in the API.)

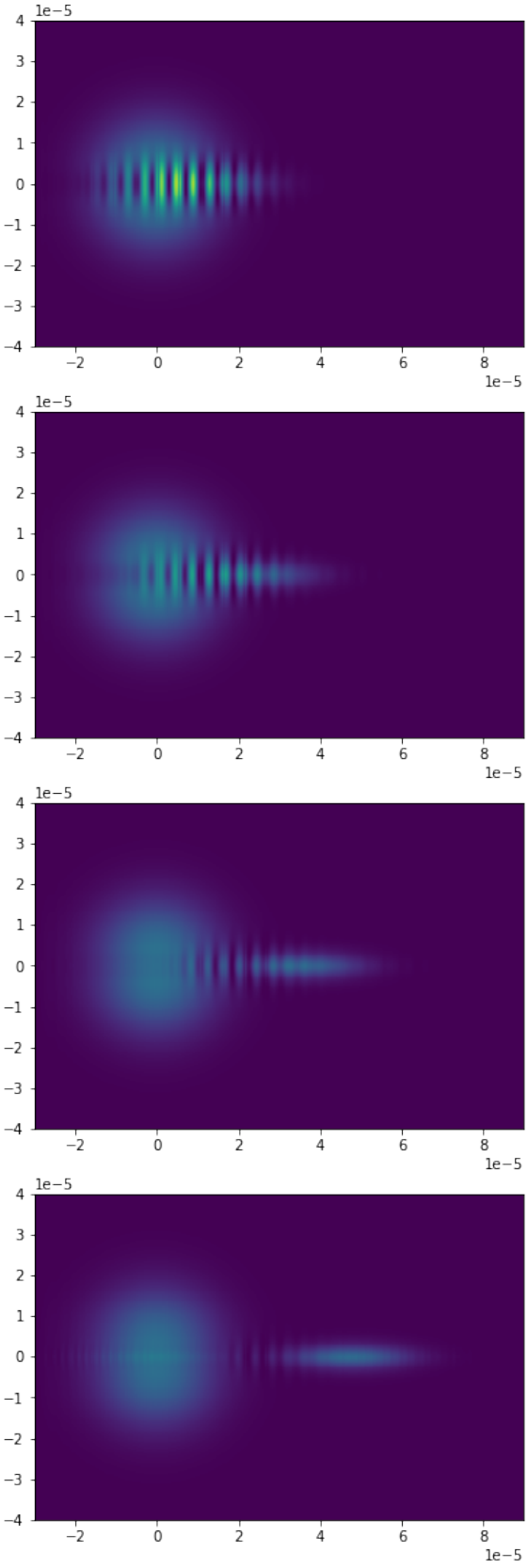
One can clearly see the interference pattern of the two counterpropagating Gaussian beams as well as their curvature. Now let us performed the Bragg pulses and see what happens to the wave function.

[59]:
```
psi.propagate(V, num_time_steps=num_time_steps,
    delta_t=pulse_length/num_time_steps,
    variables={'Omega': Omega, 'c': c, 'v': v,
    'omega_1': omega_1, 'k_1': k_1, 'w0_1': w0_1, 'zR_1': zR_1,
    'omega_2': omega_2, 'k_2': k_2, 'w0_2': w0_2, 'zR_2': zR_2,
    'Delta': Delta, 'delta': delta})

n_plots = 5
vmax = np.max(np.abs(psi.amp[:,:,0].T)**2)
fig = plt.figure(figsize=(6.4, n_plots*4.8))
for i in range(n_plots-1):
    psi.freely_propagate(num_time_steps=1, delta_t=1e-3)
```

(continues on next page)

(continued from previous page)

```
    ax = fig.add_subplot(n_plots,1,i+1, adjustable="box")
    ax.pcolormesh(psi.r[0], psi.r[1], np.abs(psi.amp[:,:,0].T)**2, rasterized=True,␣
↪vmax=vmax)
    ax.set_aspect("equal")
```

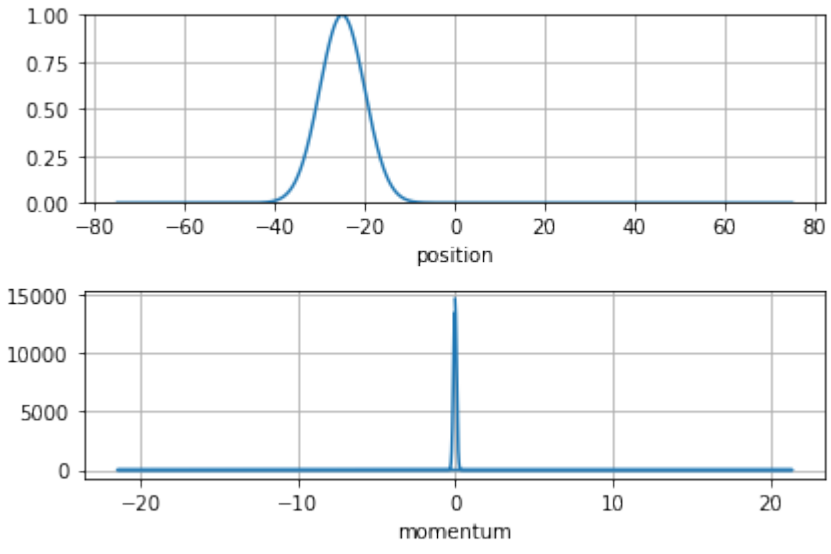

1e−5
4
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
6
8
1e−5

Since we made the Gaussian beams' waists smaller than the wave function it is only partially subjected to the Bragg transition and a big part of it stays unaffected. However, the inner part experiences the momentum kick.

2.6 Light-pulse atom interferometry with single Bragg diffraction

In this example we simulate the separation and recombination of a wave-packet that has been subjected to laser pulses inducing Bragg diffraction. In contrast to the previous example of Bragg diffraction that used the physically accurate description of a two-level system, we employ an effective description of a one-level system with a wave-packet scattered by a periodic potential resembling a crystal (hence the connection with Bragg).
We start with a gaussian wave packet offset by -25 position units and zero moneutm $k = 0$.

[60]:
```
psi = pt.Wavefunction("exp(-((x-x0)/(2*sigmax))**2)", number_of_grid_points=(1024,),
    spatial_ext=(-75,75), variables={'x0': -25, 'sigmax': 5})

def init_plot():
    fig = plt.figure()
    axs = fig.subplots(2,1)
    line1, = axs[0].plot(psi.r, np.abs(psi.amp)**2)
    axs[0].grid()
    axs[0].set_ylim(0,1)
    axs[0].set_xlabel("position")
    psi.fft();
    line2, = axs[1].plot(psi.k, np.abs(psi.amp)**2)
    axs[1].grid()
    axs[1].set_xlabel("momentum")
    psi.ifft();
    fig.tight_layout()
    return fig, axs, line1, line2

fig, axs, line1, line2 = init_plot()
```



You see the wave function in position and momentum representation. In position space it is ineed offset and in momentum space centered at 0.

We will now perform the first Bragg pulse. The effective potential is

$$V(x,t)/\hbar = 2\Omega \cos^2(k(x-kt)),$$

with Rabi frequency Ω and wave vector k. This potential drives transitions between the momentum states $|p\rangle \leftrightarrow |p+2k\rangle$. In this case we set $k=5$ and $\Omega = 2\pi$. A beamsplitter $\pi/2$ pulse is achieved after $t = \frac{\pi/2}{\Omega}$

[61]:
```
from matplotlib import animation, rc
from IPython.display import HTML

def animate_bragg_pulse(i):
    psi.propagate("2*Omega*cos(k*(x-k*t))**2", num_time_steps=1,
        delta_t=t_end/num_time_steps, variables={"Omega": 2*np.pi*f_R, "k": 5},␣
↪diag=True)
    line1.set_ydata(np.abs(psi.amp)**2)
    psi.fft()
    line2.set_ydata(np.abs(psi.amp)**2)
    psi.ifft()
    return (line1, line2)

fig, axs, line1, line2 = init_plot()

f_R = 2 # Rabi frequency in Hertz
t_end = 1/(4*f_R) # 50:50 beamsplitter
num_time_steps = 100

anim = animation.FuncAnimation(fig, animate_bragg_pulse, frames=num_time_steps,␣
↪interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[61]:
```
<IPython.core.display.HTML object>
```

As expected the pulse results in an equal superposition of momenutm states $|p\rangle$ and $|p+2k\rangle$.
In position space we already start to see interference from now two wave packets moving apart.
The wave function is now freely propagated for 5 time units such that both wave packets are seperated by $kt = 50$ position units.

[62]:
```
def animate_freeprop(i):
    psi.freely_propagate(num_time_steps=1, delta_t=t_end/num_time_steps)
    line1.set_ydata(np.abs(psi.amp)**2)
    psi.fft()
    line2.set_ydata(np.abs(psi.amp)**2)
    psi.ifft()
    return (line1, line2)

fig, axs, line1, line2 = init_plot()

t_end = 5

anim = animation.FuncAnimation(fig, animate_freeprop, frames=num_time_steps,␣
↪interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[62]:
```
<IPython.core.display.HTML object>
```

Now we will apply a mirror π pulse that will invert the momentum states in order to recombine both wave packets.

[63]:
```
fig, axs, line1, line2 = init_plot()

t_end = 1/(2*f_R) # mirror pulse

anim = animation.FuncAnimation(fig, animate_bragg_pulse, frames=num_time_steps,␣
↪interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[63]:
```
<IPython.core.display.HTML object>
```

Now the right wave packet will be in momentum state $|p\rangle$ and the left in $|p + 2k\rangle$. They propagate freely for the same time they were seperating after the first pulse.

[64]:
```
fig, axs, line1, line2 = init_plot()

t_end = 5

anim = animation.FuncAnimation(fig, animate_freeprop, frames=num_time_steps,␣
↪interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[64]:
```
<IPython.core.display.HTML object>
```

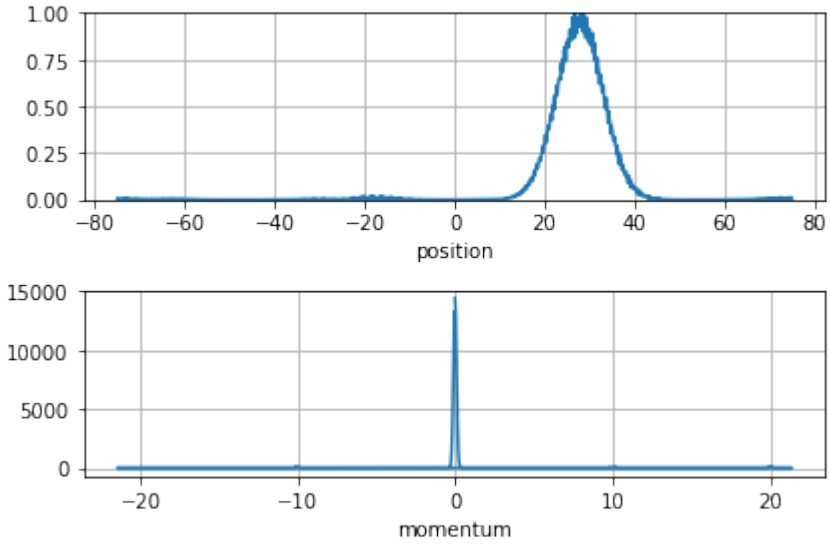
Now that they are spatially overlapping we can recombine them with a final beamsplitter pulse.

[65]:
```
fig, axs, line1, line2 = init_plot()

t_end = 1/(4*f_R) # 50:50 beamspliter pulse

anim = animation.FuncAnimation(fig, animate_bragg_pulse, frames=num_time_steps,␣
↪interval=20, blit=True)
plt.close()
HTML(anim.to_html5_video())
```

[65]:
```
<IPython.core.display.HTML object>
```

Let's have a final look at the wave function

[66]:
```
init_plot();
```



The wave function is, again, completely in its initial momentum state but has travelled the 50 position units. One can also see some interference due to imperfect beamsplitter operations to auxiliary states.

CHAPTER 3

Notes

3.1 Split-Step-Fourier method

Defining a time-propagator $U(t, t_0)$ with its action on the wave function $\Psi(t) = U(t, t_0)\Psi(t_0)$ and insert it into the Schrödinger equation, we obtain

$$i\hbar \frac{\mathrm{d}}{\mathrm{d}t} U(t, t_0)\Psi(t) = H(t)U(t, t_0)\Psi(t).$$

A solution to this is the well known Dyson series

$$U(t, t_0) = 1 - \frac{i}{\hbar}\int_{t_0}^{t} \mathrm{d}t_1 H(t_1) + \Big(-\frac{i}{\hbar}\Big)^2 \int_{t_0}^{t} \mathrm{d}t_1 \int_{t_0}^{t_1} \mathrm{d}t_2 H(t_1)H(t_2) + \cdots$$

which can be simplified using the time-sorting operator $\hat{T}$:

$$U(t, t_0) = \hat{T} \exp\Big[-\frac{i}{\hbar}\int_{t_0}^{t} H(t')\mathrm{d}t'\Big].$$

The naive intuition of the time propagator to be $\exp[-\frac{i}{\hbar}\int_{t_0}^{t} H(t')\mathrm{d}t']$, neglecting the time-sorting, is generally not true. If this naive ansatz is inserted into the Schrödinger equation one might expect the equality to hold, however, the time derivative of an exponentiated operator is only well defined if it is diagonal. Yet, generally a Hamiltonian does not commute with itself at different times $[H(t_1), H(t_2)] \neq 0$, hence they do not have the same eigenbasis and time-sorting has to be applied. Nonetheless, as an approximation for small time steps Δt, we will assume that $[H(t), H(t + \Delta t)] = 0$ holds and the naive ansatz for the time propagator can be used. (That the commutator relation $[H(t), H(t + \Delta t)] = 0$ leads to $U(t, t_0) \approx \exp[-\frac{i}{\hbar}\int_{t_0}^{t} H(t_1)\mathrm{d}t_1]$ can be even better understood using the Magnus expansion of the time propagator:

$$U(t, t_0) = 1 - \frac{i}{\hbar}\int_{t_0}^{t} H(t_1)\mathrm{d}t_1 + \frac{1}{2}\Big(-\frac{i}{\hbar}\Big)^2 \int_{t_0}^{t} \mathrm{d}t_1 \int_{t_0}^{t_1} \mathrm{d}t_2 [H(t_1), H(t_2)] + \ldots$$

The Dyson series is described, because it is the common way of writing down $U(t, t_0)$.)

Hence, we define the time evolution of a state to be approximately

$$\psi(\vec{r}, t + \Delta t) \approx e^{-\frac{i}{\hbar} H(\vec{r},t)\Delta t} \psi(\vec{r}, t).$$

The Hamiltonian consists of a kinetic T and potential V part. As it is intended to evolve the state vector incrementally over time steps of length Δt, it is necessary to calculate the exponential function of a sum of two operators. Since V is dependent on $\vec{r}$ and the kinetic part consists of the Laplacian operator, it follows (because $[V(\vec{r}), \partial^2_{\vec{r}}] \neq 0$) that for the operation $\exp[i \int (T + V)\Delta t]$ applied to a wave function, it can generally not be expressed in an appropriate eigenbasis, as both parts are not simultaneously diagonalizable.

Therefore, the Hamiltonian is split into two parts to diagonalize them separately. The potential $V(\vec{r}, t)$ is diagonal in $\vec{r}$-space, while the kinetic part is diagonal in $\vec{k}$-space as $T(\vec{k}) = -\frac{\hbar^2 \vec{k}^2}{2m}$. However, whereas for ordinary scalar quantities $e^{a+b} = e^a e^b$ holds, it does not for non-commuting operators. For this, we have to use the Lie-Trotter-Suzuki formula. This formula can be derived from the Baker–Campbell–Hausdorff relation which states

$$e^{-\frac{i}{\hbar}(T+V)\Delta t} = e^{-\frac{i}{\hbar}T\Delta t} \cdot e^{-\frac{i}{\hbar}V\Delta t} + e^{-\hbar^{-2}[T,V]\Delta t^2} + \mathcal{O}(\Delta t^3).$$

This equation would lead to accuracy of order Δt if terms with commutators would be neglected. However, the equation can be brought into the form of the Symmetric-Strang-Splitting.

$$e^{-\frac{i}{\hbar}(T+V)\Delta t} = e^{-\frac{i}{\hbar}T\frac{\Delta t}{2}} \cdot e^{-\frac{i}{\hbar}V\Delta t} \cdot e^{-\frac{i}{\hbar}T\frac{\Delta t}{2}} + \mathcal{O}(\Delta t^3)$$

which improves the accuracy by one order, without significant additional computational effort.\ Having split the Hamilton-operator into three operators, which can all be applied subsequently if the wave function is in an appropriate basis, we can evolve the wave function in time with the following algorithm:

1. The initial wave function is Fourier-transformed $\psi(\vec{r}, t) \to \mathcal{F}[\psi(\vec{r}, t)] = \psi(\vec{k}, t)$
2. The first operator is applied to the wave function $\psi(\vec{k}, t + \frac{\Delta t}{2}) = e^{i\frac{\hbar}{2m}|\vec{k}|^2 \frac{\Delta t}{2}} \psi(\vec{k}, t)$
3. Fourier transformation of ψ back into $\vec{r}$-space $\psi(\vec{k}, t + \frac{\Delta t}{2}) \to \mathcal{F}^{-1}[\psi(\vec{k}, t + \frac{\Delta t}{2})] = \psi(\vec{r}, t + \frac{\Delta t}{2})$
4. Phase correction due to potential and nonlinearity over the entire interval $\psi(\vec{r}, t + \frac{\Delta t}{2}) \to e^{-\frac{i}{\hbar}V(\vec{r}, t+\frac{\Delta t}{2})\Delta t} \psi(\vec{r}, t + \frac{\Delta t}{2})$
5. Fourier-transformation into $\vec{k}$ -space $\psi(\vec{r}, t + \frac{\Delta t}{2}) \to \mathcal{F}[\psi(\vec{r}, t + \frac{\Delta t}{2})] = \psi(\vec{k}, t + \frac{\Delta t}{2})$
6. The operator of the second kinetic half-time-step is applied $\psi(\vec{k}, t + \Delta t) = e^{i\frac{\hbar}{2m}|\vec{k}|^2 \frac{\Delta t}{2}} \psi(\vec{k}, t + \frac{\Delta t}{2})$
7. Last Fourier transformation into $\vec{r}$ -space $\psi(\vec{k}, t + \Delta t) \to \mathcal{F}^{-1}[\psi(\vec{k}, t + \Delta t)] = \psi(\vec{r}, t + \Delta t)$

Using this algorithm, the initial wave function is propagated in time $\psi(\vec{r}, t) \to \psi(\vec{r}, t + \Delta t)$, just as described by formula of the symmetric Strang splitting.

3.2 Importance of sampling frequency

The split-step Fourier method uses position and momentum space to calculate the wave-function's time-propagation. Therefore, one hast to keep in mind that the momentum state of the wave-function can only be correctly accounted for if the space-grid has enough sampling points.

This means that if Δx is the whole position space consisting of N samples, we have a sampling rate of $\delta x = \Delta x / N$. If we Fourier transform any wave-form, we can safely represent momentum states within $\Delta k = 2\pi / \delta x$. The sampling rate in momentum space is $\delta k = 2\pi / \Delta x$.

3.3 Importance of number of sampling points

Fast Fourier Transforms (FFTs) are fastet if the number of grid points of your spatial grid is a multiple of 2 e.g.: 256, 2048 etc. As every timestep involves at least two FFT we advice you to do so.

CHAPTER 4

API / Code Reference

4.1 Submodules

4.2 pytalises.wavefunction module

The Wavefunction class and its attributes.

class pytalises.wavefunction.**Wavefunction**(*psi*, *number_of_grid_points*, *spatial_ext*, *t0=0.0*, *m=1.054571817e-34*, *variables={}*, *normalize_const=None*)

Bases: `object`

Class describing wave function.

Parameters

- **psi** (*string list of strings*) – Each string describes the initial amplitude in r-space of the wave function. The number of elements is the number of internal degrees of freedom. Additional variables that are used in psi can be defined in the optional parameter variables. Predefined variables are the spatial coordinates x,y,z and time t.
- **number_of_grid_points** (*tuple of ints*) – Tuple that defines the number of grid points (nX,nY,nZ) of the wave function
- **spatial_ext** (*list of tuples*) – The supplied values define the boundary positions of the grid and thus define the actual coordinate system.
- **t0** (*float, optional*) – Internal time of wave function. Default is 0.0.
- **m** (*float, optional*) – Mass of particle described by the wavefunction. Default is 1.054571817e-34 (numerically equal to hbar).
- **variables** (*dict*) – Dictionary of additionally used variables in the definition of the wave function in psi. Predefined variables are the spatial coordinates x,y,z and time t.

- **normalize_const** (*float, optional*) – Normalizes the wave function such that the integral of |Psi|^2 over all internal and external degrees of freedom equals *normalize_const*

num_int_dim
The number of internal degrees of freedom

Type int

num_ex_dim
The number of external degrees of freedom

Type int

r
The arrays are the evenly spaced spatial coordinates as defined through definition of spatial_ext and number_of_grid_points

Type list of 1d arrays

k
The arrays are the evenly spaced inverse spatial coordinates as defined through definition of spatial_ext and number_of_grid_points

Type list of 1d arrays

Examples

Wavefunction with two internal states where the first state is gaussian distributed in 1d r-space and the second state is not occupied at all.

```
>>> from pytalises import Wavefunction
>>> psi = Wavefunction(["exp(-((x-x0)/a0)**2)", "0.0"],
    (16,), [(-2,2),], variables={'a0':1/2, 'x0':0})
>>> print(psi.num_int_dim)
2
>>> print(psi.num_ext_dim)
1
>>> print(psi.amp)
[[1.12535175e-07+0.j 0.00000000e+00+0.j]
[6.03594712e-06+0.j 0.00000000e+00+0.j]
[1.83289361e-04+0.j 0.00000000e+00+0.j]
[3.15111160e-03+0.j 0.00000000e+00+0.j]
[3.06707930e-02+0.j 0.00000000e+00+0.j]
[1.69013315e-01+0.j 0.00000000e+00+0.j]
[5.27292424e-01+0.j 0.00000000e+00+0.j]
[9.31358402e-01+0.j 0.00000000e+00+0.j]
[9.31358402e-01+0.j 0.00000000e+00+0.j]
[5.27292424e-01+0.j 0.00000000e+00+0.j]
[1.69013315e-01+0.j 0.00000000e+00+0.j]
[3.06707930e-02+0.j 0.00000000e+00+0.j]
[3.15111160e-03+0.j 0.00000000e+00+0.j]
[1.83289361e-04+0.j 0.00000000e+00+0.j]
[6.03594712e-06+0.j 0.00000000e+00+0.j]
[1.12535175e-07+0.j 0.00000000e+00+0.j]]
>>> print(psi.r)
[array([-2.        , -1.73333333, -1.46666667, -1.2       , -0.93333333,
   -0.66666667, -0.4       , -0.13333333,  0.13333333,  0.4       ,
```

(continues on next page)

(continued from previous page)

```
    0.66666667,  0.93333333,  1.2       ,  1.46666667,  1.73333333,
    2.        ]), array([0.]), array([0.])]
```

amp
Ndarray of the wave function amplitudes.

construct_FFT(*num_of_threads=1*, *FFTWflags=('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT')*)
Construct pyfftw bindings.

exp_pos(*axis=None*)
Calculate the expected position on given axis.

Calculates the mean position of Psi on chosen axis. Axes 0,1,2 correspond to x,y,z. The other two axes are traced out. If no axis iv given returns array of mean position of all external degrees of freedom.

freely_propagate(*num_time_steps*, *delta_t*, *num_of_threads=1*, *FFTWflags=('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT')*)
Propagates the Wavefunction object in time with V=0.

Function that can propagate the wavefunction if no potential is present.

Parameters

- **num_time_steps** (*int*) – Number of times the wavefunction is propagated by time delta_t using the Split-Step Fourier method.
- **delta_t** (*float*) – Time increment the wavefunction is propagated in one time step.
- **num_of_threads** (*int, optional*) – Number of threads uses for calculation. Default is 1.
- **FFTWflags** (*tuple of strings*) – Options for FFTW planning [1]. Default is ('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT',).

References

[1] http://www.fftw.org/fftw3_doc/Planner-Flags.html

k
(list of) array of the wave function position axes.

normalize_to(*n_const*)
Normalize the wave function.

Normalizes the wave function such that the integral of |Psi|^2 over all internal and external states equals n_const

propagate(*potential*, *num_time_steps*, *delta_t*, ***kwargs*)
Propagates the Wavefunction object in time.

Function that propagates the wavefunction using a Split-Step Fourier method [1].

Parameters

- **potential** (*string list of strings*) – This list contains the matrix elements of the potential term V in string format. If the potential has nondiagonal elements (see optional parameter diag) each elements represents one matrix element of the lower triangular part of V. For example a 3x3 potential with nondiagonal elements would be of form potential=[H00, H10, H20, H11, H21, H22]. If the potential term is supposed to have only

diagonal elements (diag=True), the potential parameter for a 3x3 potential would look like potential=[H00,H11,H22].

- **num_time_steps** (*int*) – Number of times the wavefunction is propagated by time delta_t using the Split-Step Fourier method.
- **delta_t** (*float*) – Time increment the wavefunction is propagated in one time step.
- **variables** (*dict, optional*) – Dictionary containing values for variables you might have used in potential
- **diag** (*bool , optional*) – If true, no numerical diagonalization has to be invoked in order to calculate time-propagation as nondiagonal elements are omitted. This makes the computation much faster. Default is False.
- **num_of_threads** (*int, optional*) – Number of threads uses for calculation. Default behaviour is to use all threads available.
- **FFTWflags** (*tuple of strings*) – Options for FFTW planning [2]. Default is ('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT',).

References

[1] https://en.wikipedia.org/wiki/Split-step_method [2] http://www.fftw.org/fftw3_doc/Planner-Flags.html

r
(list of) array of the wave function position axes.

state_occupation(*nth_state=None*)
Return occupation number of nth internal state.

Evaluates the spatial integral over |Psi|^2 for the nth internal state. If none is given a vector of the occupation number of all internal states is returned.

var_pos(*axis=None*)
Calculate the variance on given axis.

Calculates the variance in position of Psi on chosen axis. Axes 0,1,2 correspond to x,y,z. The other two axes are traced out. If no axis iv given returns array of variance position of all external degrees of freedom.

pytalises.wavefunction.**assert_type_or_list_of_type**(*argument*, *wished_type*)

4.3 pytalises.propagator module

Module containing functions that help propagating the Wavefunction class.

class pytalises.propagator.**Propagator**(*psi*, *potential*, *variables={}*, *diag=False*, *num_of_threads=1*, *FFTWflags=('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT')*)

Bases: object

Class for propagating instances of the Wavefunction class.

Parameters

- **psi** (Wavefunction) – The Wavefunction object the Propagator class acts on

- **potential** (*list of strings*) – This list contains the matrix elements of the potential term V in string format. If the potential has nondiagonal elements (see optional parameter diag) each elements represents one matrix element of the lower triangular part of V. For example a 3x3 potential with nondiagonal elements would be of form potential=[H00, H10, H20, H11, H21, H22]. If the potential term is supposed to have only diagonal elements (diag=True), the potential argument for a 3x3 potential would look like potential=[H00,H11,H22].
- **variables** (*dict, optional*) – Dictionary containing values for variables you might have used in potential
- **diag** (*bool , optional*) – If true, no numerical diagonalization has to be invoked in order to calculate time-propagation. Default is False.
- **num_of_threads** (*int, optional*) – Number of threads uses for calculation. Default is 1.
- **FFTWflags** (*tuple of strings*) – Options for FFTW planning [1]. Default is ('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT',).

References

[1] http://www.fftw.org/fftw3_doc/Planner-Flags.html

class Potential(*potential_string*, *variables={}*, *diag=False*)
Bases: object

Simple class for collecting information about the potential.

diag_potential_prop(*delta_t*)
Calculate exp(i*V/hbar*delta_t)*Psi by simple matrix multiplication.

This method is used if the potential matrix V is diagonal. This is much faster than *nondiag_potential_prop* and should be used if possible.

eval_V()
Evalutes V on the whole spatial grid.

The result is saved in Propagator.V_eval_array.

eval_diag_V()
Evalutes diagonal elements of V on the whole spatial grid.

The result is saved in Propagator.V_eval_array.

kinetic_prop(*delta_t*)
Perform time propagation in k-space.

Transforms the Wavefunction into k-space, calculates exp(i*hbar/(2m)*k**2*delta_t)*Psi(kx,ky,kz) and transforms it back into r-space.

nondiag_potential_prop(*delta_t*)
Calculate exp(i*V/hbar*delta_t)*Psi using numerical diagonalization.

This method has to be used if the potential matrix has nondiagonal elements.

potential_prop(*delta_t*)
Wrap function that calculates exp(i*V(x,y,z)/hbar*delta_t)*Psi(x,y,z).

This can be either nondiag_potential_prop or diag_potential_prop.

pytalises.propagator.**freely_propagate**(*psi*, *num_time_steps*, *delta_t*, *num_of_threads=1*, *FFTWflags=('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT')*)

Propagates a Wavefunction object in time with V=0.

Function that can propagate the wavefunction if no potential is present.

Parameters

- **psi** (Wavefunction) – The Wavefunction object the Propagator class acts on
- **num_time_steps** (*int*) – Number of times the wavefunction is propagated by time delta_t using the Split-Step Fourier method.
- **delta_t** (*float*) – Time increment the wavefunction is propagated in one time step.
- **num_of_threads** (*int, optional*) – Number of threads uses for calculation. Default is 1.
- **FFTWflags** (*tuple of strings*) – Options for FFTW planning [1]. Default is ('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT',).

References

[1] http://www.fftw.org/fftw3_doc/Planner-Flags.html

pytalises.propagator.**get_eig**

Calculate eigenvectors and eigenvalues of matrices in array.

JIT-compiled function that calculates the eigenvectors and eigenvalues of input array M in parallel using numba. The resulting eigenvectors are stored in the input matrix and the eigenvalues in the array eigvals.

Parameters

- **M**(*3d array of (NxN) arrays*) –
- **eigvals**(*3d array of 1d arrays with N elements*) –

pytalises.propagator.**propagate**(*psi*, *potential*, *num_time_steps*, *delta_t*, ***kwargs*)

Propagates a Wavefunction object in time.

Function that propagates the wavefunction using a Split-Step Fourier method [1].

Parameters

- **psi** (Wavefunction) – The Wavefunction object the Propagator class acts on
- **potential** (*string list of strings*) – This list contains the matrix elements of the potential term V in string format. If the potential has nondiagonal elements (see optional parameter diag) each elements represents one matrix element of the lower triangular part of V. For example a 3x3 potential with nondiagonal elements would be of form potential=[H00, H10, H20, H11, H21, H22]. If the potential term is supposed to have only diagonal elements (diag=True), the potential parameter for a 3x3 potential would look like potential=[H00,H11,H22].
- **num_time_steps** (*int*) – Number of times the wavefunction is propagated by time delta_t using the Split-Step Fourier method.
- **delta_t** (*float*) – Time increment the wavefunction is propagated in one time step.
- **variables** (*dict, optional*) – Dictionary containing values for variables you might have used in potential

- **diag** (*bool , optional*) – If true, no numerical diagonalization has to be invoked in order to calculate time-propagation as nondiagonal elements are omitted. This makes the computation much faster. Default is False.
- **num_of_threads** (*int, optional*) – Number of threads uses for calculation. Default behaviour is to use all threads available.
- **FFTWflags** (*tuple of strings*) – Options for FFTW planning [2]. Default is ('FFTW_ESTIMATE', 'FFTW_DESTROY_INPUT',).

References

[1] https://en.wikipedia.org/wiki/Split-step_method [2] http://www.fftw.org/fftw3_doc/Planner-Flags.html

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index