A Tutorial for iDEA - interacting Dynamic Electrons Approach

In this tutorial we will introduce you to the main functionality of iDEA. For more details, please see the full API documentation.

iDEA allows you to describe quantum mechanical systems in one dimension, and solve them for the required quantum state, from which you can calculate observables. The method to solve the system can be both exact, or a variaty of common and novel approximate methods. These states can then be propagated due to perturbations, and such evolutions can be invesigated. This allows you to both gain key insights from the exact solution, and by investigating failings of existing methods, investigate why such failings occur, and how to create new more accurate methods. In addition, iDEA can be used to create interactive visualisations of concepts from quantum mechanics and condenced matter physics, to create engaging teaching material. Also, the iDEA code, and its minimal dependencies are composed of fully free software and are highly optimised, and so can allow researchers to release notebooks with thier papers, ensuring papers can be reproduces at the touch of a button. This also increases the extent to which readers can engaage with your results.

This tutorial will follow the following outline:

1. The "Hello World" of iDEA.
2. Describing Quantum Systems.
3. Solving Systems Exactly.
4. Computing Observables.
5. Perturbations and Time-Dependence.
6. Approximate Methods.
7. Reverse Engineering.
8. A full example: Investigating the 1D hydrogen molecule.

To install iDEA, simply use pip: `pip install iDEA-latest` .

Now we can import our depencies for this tutorial:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import iDEA
```

1. The "Hello World" of iDEA.

When learning a new tool, it is always useful to try the simplest possible example.

In this case, we will use iDEA to determine **the ground-state charge density of the 2 electron atom in one-dimension.**

Firstly, iDEA has this as a built-in pre-defined system. Let us take this, and print it:

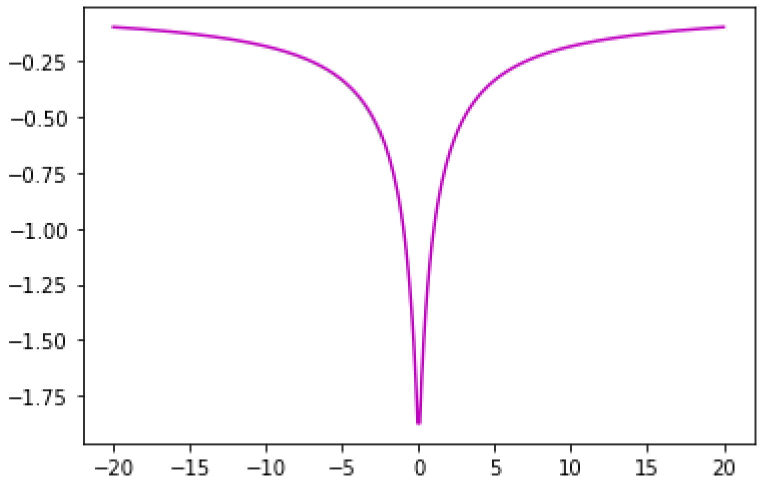
In [2]:
```
atom = iDEA.system.systems.atom
print(atom)
```

```
iDEA.system.System: x = np.array([-20.000,...,20.000]), dx = 0.1338..., v_ext
= np.array([-0.095,...,-0.095]), electrons = ud
```

As we can see this system contains two electrons, one up-spin and one down-spin, as shown by `electrons = ud` . And it exists on a one-dimensional grid from $x=-20$ to $x=20$, with a grid spacing of $dx = 0.1338...$.

Let's plot the atom's external potential $V_{\mathrm{ext}}(x)$:

In [3]:
```
plt.plot(atom.x, atom.v_ext, 'm-')
plt.show()
```



This represents a softened Coloumb interaction to two protons located at $x=0$.

Now we have the atom described by an external potential, and containing two electrons, let use solve the system for its **ground-state** `(k=0)` to give the many-body wavefunction, which fully describes the quantum-mechanical system.

Since we want the exact solution, we will use the many-body `interacting` method, which treats the electron interaction exactly:

In [4]:
```
ground_state = iDEA.methods.interacting.solve(atom, k=0)
```

This object contains the wavefunction of the state.

We can use the help function to see what this object contains.

In [5]:
```
help(ground_state)
```

```
Help on ManyBodyState in module iDEA.state object:

class ManyBodyState(State)
 |  ManyBodyState(space: numpy.ndarray = None, spin: numpy.ndarray = None, fu
ll=None, energy=None)
 |
 |  State of interacting particles.
 |
 |  Method resolution order:
 |      ManyBodyState
 |      State
 |      abc.ABC
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, space: numpy.ndarray = None, spin: numpy.ndarray = None, f
ull=None, energy=None)
 |      State of particles in a many-body state.
 |      
 |      This is described by a spatial part
 |      .. math:: \psi(x_1,x_2,\dots,x_N)
 |      on the spatial grid, and a spin
 |      part on the spin grid
 |      .. math:: \chi(\sigma_1,\sigma_2,\dots,\sigma_N).
 |      These are NOT necessarily antisymmetric states,
 |      they can be combined using the antisymmetrisation operaration to prod
uce the full
 |      wavefunction
 |      .. math:: \Psi(x_1,\sigma_1,x_2,\sigma_2,\dots,x_N,\sigma_N).
 |      
 |      | Args:
 |      |     space: np.ndarray, Spatial part of the wavefunction on the spat
ial grid \psi(x_1,x_2,\dots,x_N). (default = None)
 |      |     spin: np.ndarray, Spin part of the wavefunction on the spin gri
d \chi(\sigma_1,\sigma_2,\dots,\sigma_N). (default = None)
 |      |     full: np.ndarray, Total antisymmetrised wavefunction \Psi(x_
1,\sigma_1,x_2,\sigma_2,\dots,x_N,\sigma_N). (default = None)
 |      |     energy: float, Total energy of the state.
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  __abstractmethods__ = frozenset()
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from State:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
```

As we can see, it contains three arrays: the space, spin and fully antisymmetrised wavefunction:

In [6]:
```python
print(ground_state.space.shape)
print(ground_state.spin.shape)
print(ground_state.full.shape)
```

```
(300, 300)
(2, 2)
(300, 2, 300, 2)
```

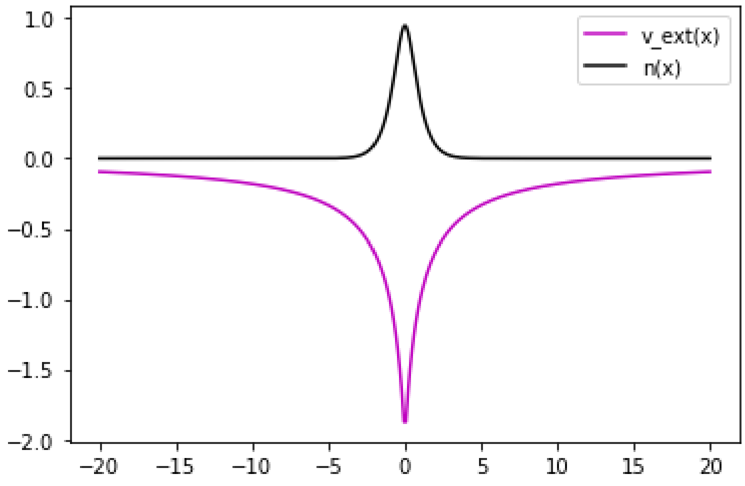
As there are 300 grid points in our system, and 2 possibilities for spin, these shapes are what we expect.

We would now like to calculate our observable of interest: the charge density for the atom in it's ground-state. This is simply done as follows:

In [7]:
```
n = iDEA.observables.density(atom, state=ground_state)
```

Let's add that to our plot:

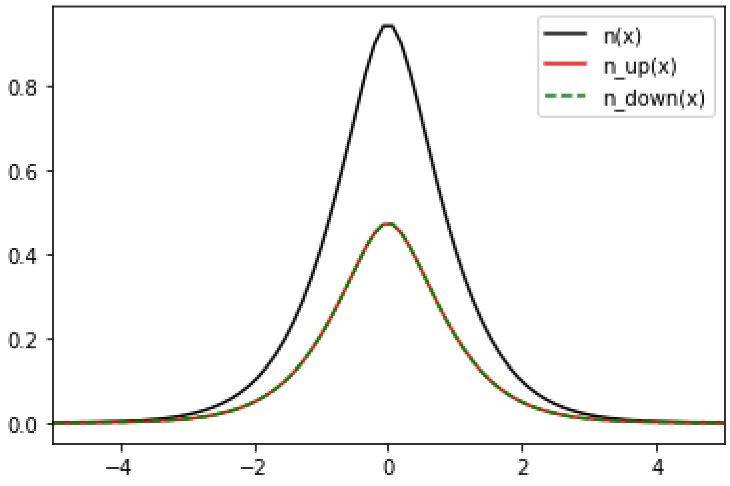
In [8]:
```
plt.plot(atom.x, atom.v_ext, 'm-', label='v_ext(x)')
plt.plot(atom.x, n, 'k-', label='n(x)')
plt.legend()
plt.show()
```



We can also compute the up and down spin-densities seperatly, and of course they are the same for this system:

In [9]:
```
n, up_n, down_n = iDEA.observables.density(atom, state=ground_state, return_s
```

In [10]:
```
plt.plot(atom.x, n, 'k-', label='n(x)')
plt.plot(atom.x, up_n, 'r-', label='n_up(x)')
plt.plot(atom.x, down_n, 'g--', label='n_down(x)')
plt.xlim([-5.0, 5.0])
plt.legend()
plt.show()
```



There are many more observables we could compute. To see the full listing of any iDEA module, use the `dir` function:

In [11]:
```
dir(iDEA.observables)
```

Out[11]:
```
['Union',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_placeholder',
 'copy',
 'density',
 'density_matrix',
 'exchange_energy',
 'exchange_potential',
 'external_energy',
 'external_potential',
 'hartree_energy',
 'hartree_potential',
 'iDEA',
 'itertools',
 'kinetic_energy',
 'np',
 'observable',
 'single_particle_energy',
 'string']
```

If you would like to analyse the source code of any particular part of iDEA, we can use `inspect`:

In [12]:
```
import inspect
print(inspect.getsource(iDEA.observables.hartree_potential))
```

```
def hartree_potential(s: iDEA.system.System, n: np.ndarray) -> np.ndarray:
    r"""
    Compute the Hartree potential from a density.

    | Args:
    |     s: iDEA.system.System, System object.
    |     n: np.ndarray, Charge density of the system.

    | Returns:
    |     v_h: np.ndarray, Hartree potential, or evolution of Hartree potenti
al.
    """
    if len(n.shape) == 1:
        v_h = np.dot(n, s.v_int) * s.dx
        return v_h

    elif len(n.shape) == 2:
        v_h = np.zeros_like(n)
        for j in range(v_h.shape[0]):
            v_h[j, :] = np.dot(n[j, :], s.v_int[:, :]) * s.dx
        return v_h

    else:
        raise AttributeError(f"Expected array of shape 1 or 2, got {n.shape}
instead.")
```

2. Describing Quantum Systems.

In the simple example in the last section we used a pre-defined system. In this section, we will show you how to define any custom 1D quantum system.

In iDEA, a quantum system is defined by:

- A spatial grid.
- An external potential.
- An electronic interaction.
- A given number of up and down spin electrons.

We can see the details using the `help` function:

In [13]:
```
help(iDEA.system.System)
```

```
Help on class System in module iDEA.system:

class System(builtins.object)
 |  System(x: numpy.ndarray, v_ext: numpy.ndarray, v_int: numpy.ndarray, elec
trons: str, stencil: int = 13)
 |  
 |  Model system, containing all defining properties.
 |  
 |  Methods defined here:
 |  
 |  __init__(self, x: numpy.ndarray, v_ext: numpy.ndarray, v_int: numpy.ndarr
ay, electrons: str, stencil: int = 13)
 |      Model system, containing all defining properties.
 |      
 |      | Args:
 |      |     x: np.ndarray, Grid of x values in 1D space.
 |      |     v_ext: np.ndarray, External potential on the grid of x values.
 |      |     v_int: np.ndarray, Interaction potential on the grid of x value
s.
 |      |     electrons: string, Electrons contained in the system.
 |      |     stencil: int, Stencil to use for derivatives on the grid of x v
alues. (default = 13)
 |      
 |      | Raises:
 |      |     AssertionError.
 |  
 |  __str__(self)
 |      Return str(self).
 |  
 |  check(self)
 |      Performs checks on system properties. Raises AssertionError if any ch
eck fails.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  dx
 |
 |  electrons
 |
 |  x
```

Let us define a system `s` , which contains wo electrons (both spin-up) in a quantum harmonic oscillator (QHO).

First, let us define a x-grid from $x=-10$ to $x=10$ with 250 grid points:

In [14]:
```
x = np.linspace(-10, 10, 150)
```

Now we can define the external potential of the QHO:

$V_\mathrm{ext}(x) = \frac{1}{2} \omega^2 x^2$ where $\omega=0.25$

In [15]:
```
v_ext = 0.5 * 0.25**2 * x**2
```

Now we will choose an interaction for our electrons on our grid. As this is 1D we choose the softened interaction, but other options are available *(including custom interactions to build hubbard systems.)*

In [16]:
```
v_int = iDEA.interactions.softened_interaction(x)
```

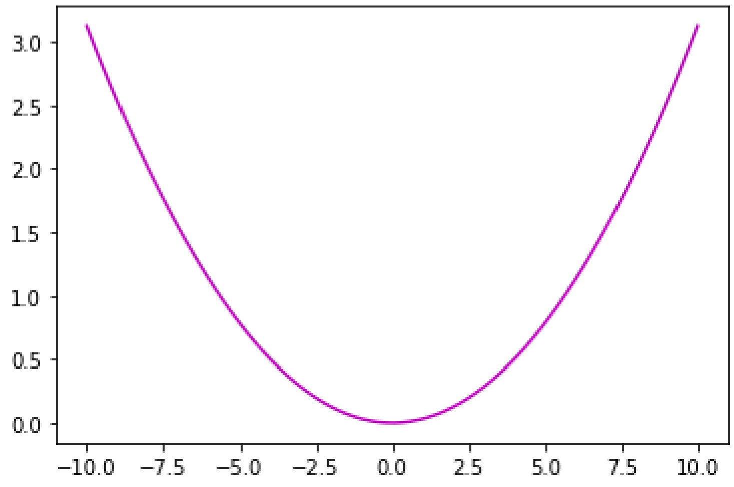
We can now build our system:

In [17]:
```
s = iDEA.system.System(x, v_ext, v_int, electrons='uu')
print(s)
print("number of electrons =", s.count)
```

```
iDEA.system.System: x = np.array([-10.000,...,10.000]), dx = 0.1342..., v_ext
= np.array([3.125,...,3.125]), electrons = uu
number of electrons = 2
```

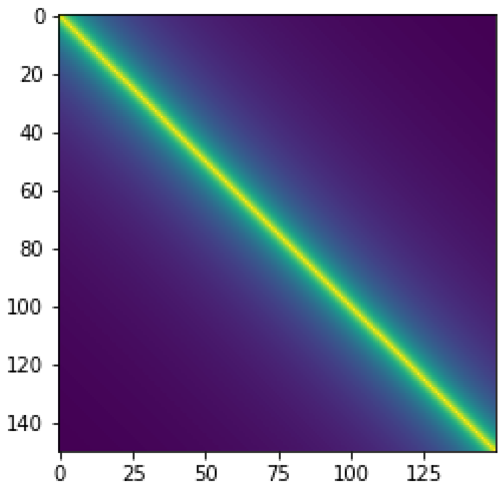
We can now plot our external potential:

In [18]:
```
plt.plot(s.x, s.v_ext, 'm-')
plt.show()
```



And our interaction $V_\mathrm{int}(x,x')$:

In [19]:
```
plt.imshow(v_int)
plt.show()
```



And to summerise, the code we used to build our system:

In [20]:
```
x = np.linspace(-10, 10, 150)
v_ext = 0.5 * 0.25**2 * x**2
v_int = iDEA.interactions.softened_interaction(x)
s = iDEA.system.System(x, v_ext, v_int, electrons='uu')
```

3. Solving Quantum Systems Exacly.

Now we have the description of our quantum system, we can use iDEA to solve the Schrödinger equation for the ground-state (`k=0`) of this system using the `interacting` module:

In [21]:
```
ground_state = iDEA.methods.interacting.solve(s, k=0)
```

We can now use the spatial, spin and full wavefunctions to do any analysis we need:

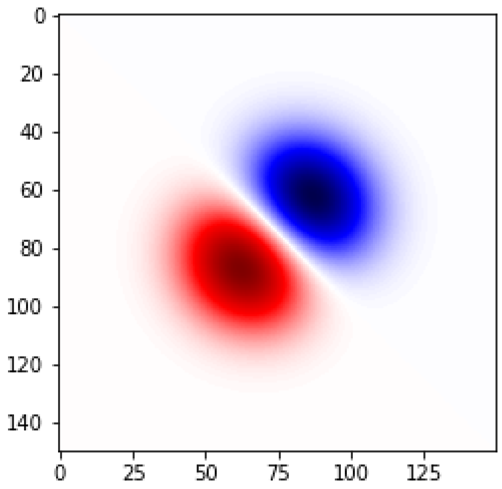
In [22]:
```
print(ground_state.space.shape)
print(ground_state.spin.shape)
print(ground_state.full.shape)
```

```
(150, 150)
```

```
(2, 2)
(150, 2, 150, 2)
```

Since this is a two-electron system, we can actually plot the spacial part of the wavefunction directly in 2D:

```
In [23]:
plt.imshow(ground_state.space.real, cmap="seismic", vmax=np.max(ground_state.
plt.show()
```



Note that this is zero along the diagonal, owing to the Pauli exclusion principle. (As an excerise, show this is not zero if the electrons have opposise spin. Hint: when building the system use `electrons = "ud"` insead of `"uu"`)

We can test that the wavefunction is antisymmetrised:

```
In [24]:
np.allclose(ground_state.full.real[80,0,50,0], -ground_state.full.real[50,0,8
```

```
Out[24]: True
```

From this object we can also get the total energy:

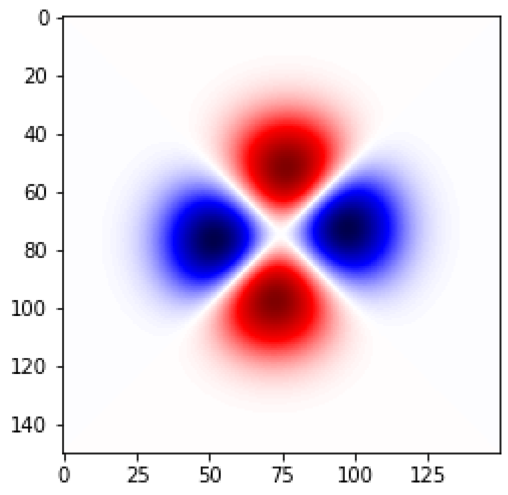
```
In [25]:
print(ground_state.energy)
```

```
0.753177944467693
```

We can also compute **any excited state** we would like. For example, the first excited state:

```
In [26]:
first_state = iDEA.methods.interacting.solve(s, k=1)
```

```
In [27]:
plt.imshow(first_state.space.real, cmap="seismic", vmax=np.max(first_state.sp
plt.show()
```



We could use this to compute the **optical gap** of the system:

In [28]:

```
print("optical gap =", first_state.energy - ground_state.energy)
```

```
optical gap = 0.2500000395902423
```

4. Computing Observables.

Let us now turn to computing some common observables of our system, and visualising them:

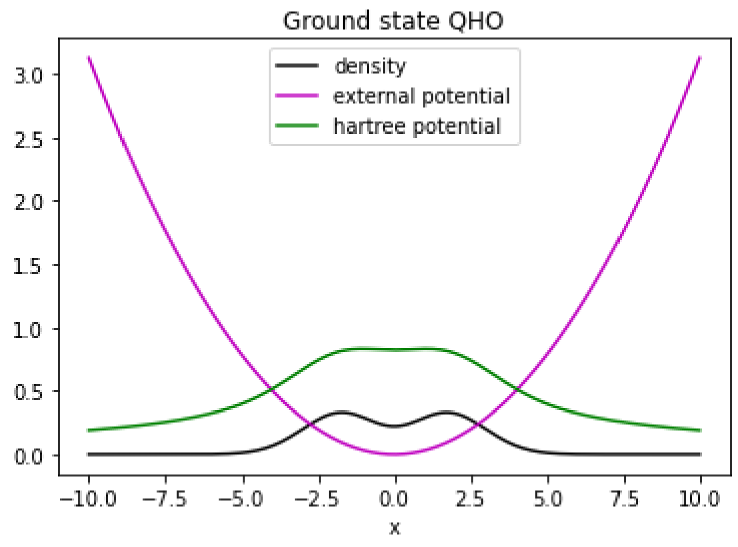
In [29]:

```
n = iDEA.observables.density(s, state=ground_state)
v_h = iDEA.observables.hartree_potential(s, n)
E_h = iDEA.observables.hartree_energy(s, n, v_h)


p = iDEA.observables.density_matrix(s, state=ground_state)
v_x = iDEA.observables.exchange_potential(s, p)
E_x = iDEA.observables.exchange_energy(s, p, v_x)


print("hartree energy =", E_h, ", exchange energy =", E_x)
```

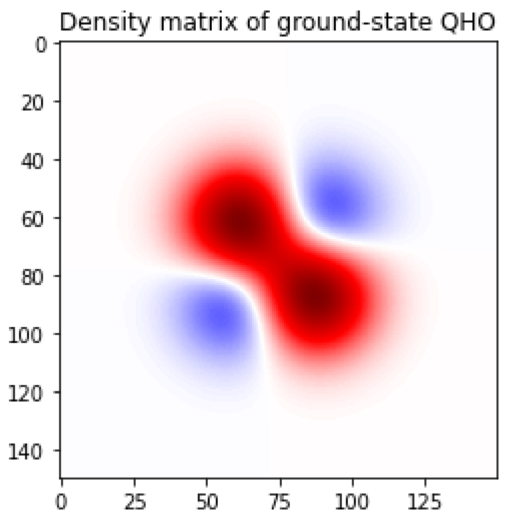
```
hartree energy = 0.7585662970835491 , exchange energy = -0.5146141349952208
```

In [30]:

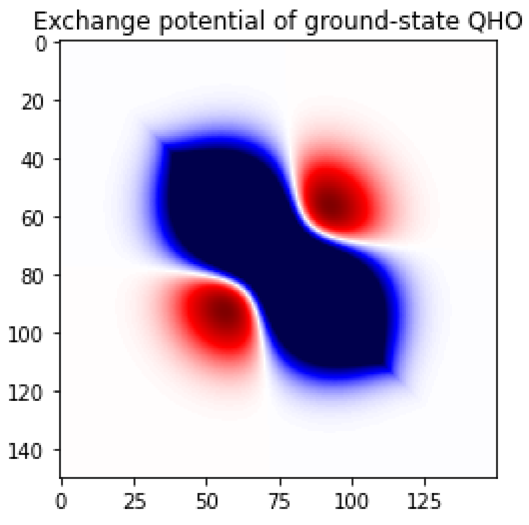
```
plt.plot(s.x, n, 'k-', label="density")
plt.plot(s.x, s.v_ext, 'm-', label="external potential")
plt.plot(s.x, v_h, 'g-', label="hartree potential")
plt.legend()
plt.xlabel("x")
plt.title("Ground state QHO")
plt.show()
```



In [31]:
```
plt.imshow(p, cmap="seismic", vmax=np.max(p), vmin=-np.max(p))
plt.title("Density matrix of ground-state QHO")
plt.show()
```



In [32]:
```
plt.imshow(v_x, cmap="seismic", vmax=np.max(v_x), vmin=-np.max(v_x))
plt.title("Exchange potential of ground-state QHO")
plt.show()
```

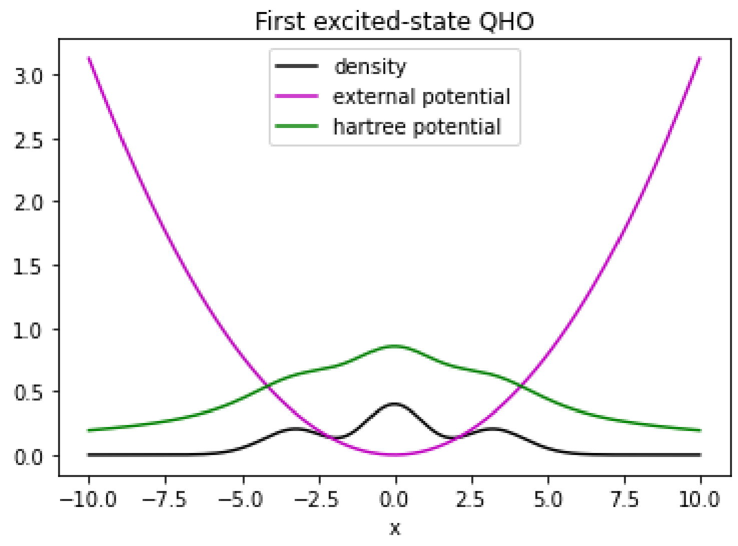


And we can of course repeat the above for the first excited-state:

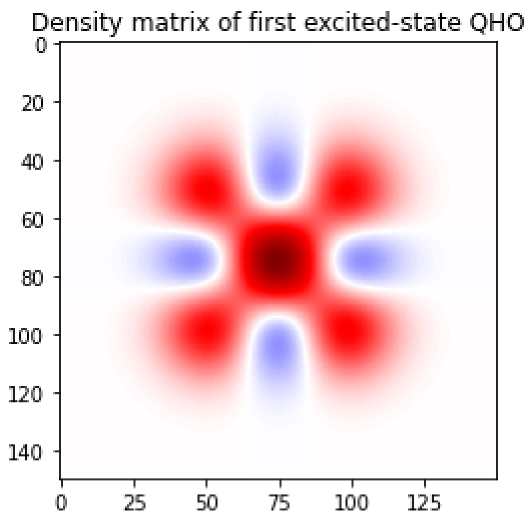
In [33]:
```
n = iDEA.observables.density(s, state=first_state)
v_h = iDEA.observables.hartree_potential(s, n)
E_h = iDEA.observables.hartree_energy(s, n, v_h)


p = iDEA.observables.density_matrix(s, state=first_state)
v_x = iDEA.observables.exchange_potential(s, p)
E_x = iDEA.observables.exchange_energy(s, p, v_x)


print("hartree energy =", E_h, ", exchange energy =", E_x)
```

```
hartree energy = 0.6990259617571023 , exchange energy = -0.4397489413446015
```

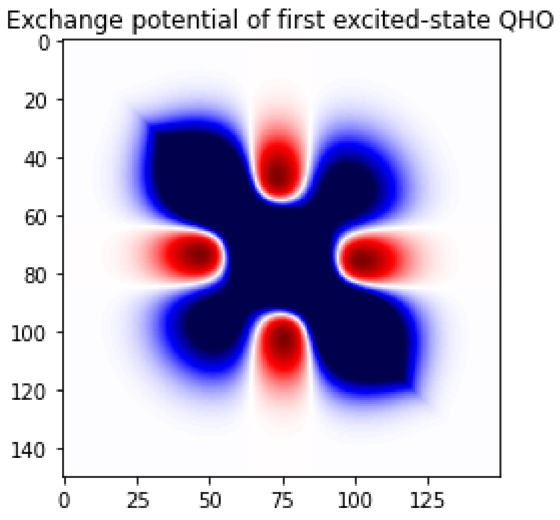
In [34]:
```
plt.plot(s.x, n, 'k-', label="density")
plt.plot(s.x, s.v_ext, 'm-', label="external potential")
plt.plot(s.x, v_h, 'g-', label="hartree potential")
plt.legend()
plt.xlabel("x")
plt.title("First excited-state QHO")
plt.show()
```



In [35]:
```
plt.imshow(p, cmap="seismic", vmax=np.max(p), vmin=-np.max(p))
plt.title("Density matrix of first excited-state QHO")
plt.show()
```



In [36]:
```
plt.imshow(v_x, cmap="seismic", vmax=np.max(v_x), vmin=-np.max(v_x))
plt.title("Exchange potential of first excited-state QHO")
plt.show()
```



5. Perturbations and Time-Dependence.

Now we know how to; describe systems, solve them for given states, and compute observables, we can now look at driving these systems out of thier stationary states using a external perturbation.

First we define a time grid $t=0 \rightarrow t=100$: wih 1000 timesteps:

In [37]:
```
t = np.linspace(0, 50, 500)
```

For our perturbation, we choose a oscillating linear electric field:

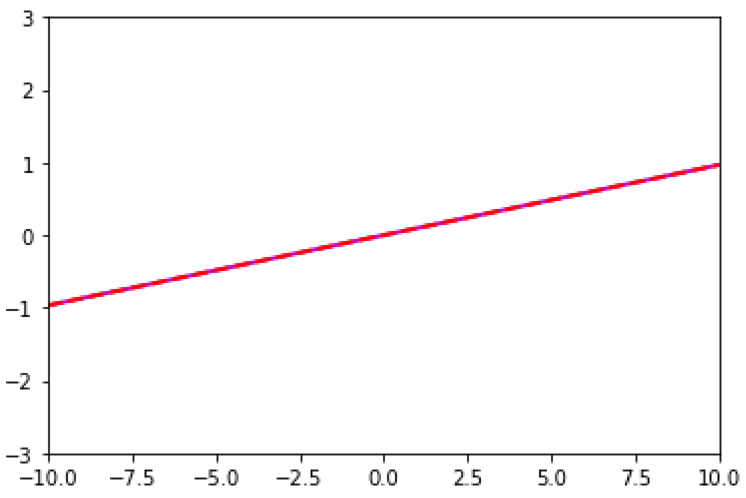
In [38]:
```
v_ptrb = np.zeros(shape=t.shape+x.shape)
for j, ti in enumerate(t):
    v_ptrb[j,:] = -0.1 * x * np.sin(0.5*ti)
```

We can animate our field:

In [39]:
```
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
fig = plt.figure()
ax = plt.axes(xlim=(s.x[0], s.x[-1]), ylim=(-3.0, 3.0))
line1, = ax.plot([], [], lw=2, c='m')
line2, = ax.plot([], [], lw=2, c='r', ls='--')
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    return line1, line2
def animate(i):
    line1.set_data(s.x, v_ptrb[i,:])
    line2.set_data(s.x, v_ptrb[i,:])
    return line1, line2
anim = FuncAnimation(fig, animate, init_func=init, frames=100, interval=10, b
HTML(anim.to_html5_video())
```

Out[39]:

0:00 / 0:01



To propagate our ground-state, due to this perturbation: *(this can take some time on slower machine, to speed up simply reduce the number of timesteps)*

In [40]:
```
evolution = iDEA.methods.interacting.propagate(s, ground_state, v_ptrb, t)
```

```
iDEA.methods.interacting.propagate:
```

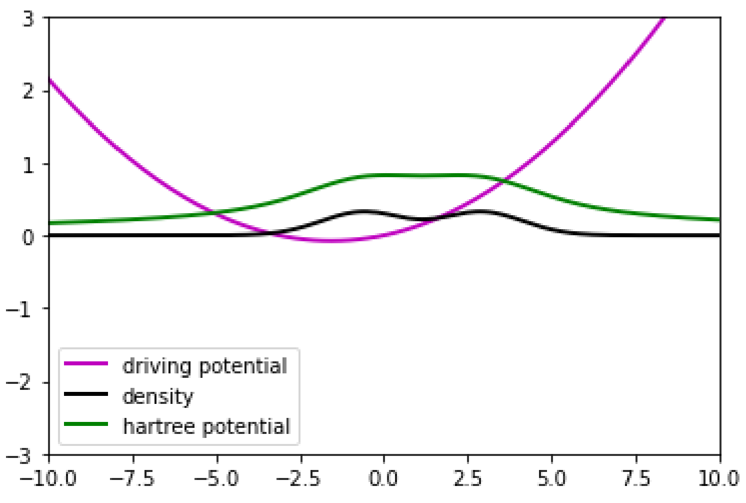
From this evolution, we can compute dynamic observables in much the same way as previously.

In [41]:
```
n = iDEA.observables.density(s, evolution=evolution)
v_h = iDEA.observables.hartree_potential(s, n)
E_h = iDEA.observables.hartree_energy(s, n, v_h)


p = iDEA.observables.density_matrix(s, evolution=evolution)
v_x = iDEA.observables.exchange_potential(s, p)
E_x = iDEA.observables.exchange_energy(s, p, v_x)
```

In [42]:
```
fig = plt.figure()
ax = plt.axes(xlim=(s.x[0], s.x[-1]), ylim=(-3.0, 3.0))
line1, = ax.plot([], [], lw=2, c='m', label="driving potential")
line2, = ax.plot([], [], lw=2, c='k', label="density")
line3, = ax.plot([], [], lw=2, c='g', label="hartree potential")
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    line3.set_data([], [])
    return line1, line2, line3
def animate(i):
    line1.set_data(s.x, s.v_ext[:] + v_ptrb[i,:])
    line2.set_data(s.x, n[i,:])
    line3.set_data(s.x, v_h[i,:])
    return line1, line2, line3
plt.legend()
anim = FuncAnimation(fig, animate, init_func=init, frames=100, interval=10, b
HTML(anim.to_html5_video())
```

Out[42]:

0:00 / 0:01



6. Approximate Methods.

As well as the exact solution, we can also compute everything we have in the previous sections, but with common approximations. To see the current list of approximations (more are added each release!) we can use the `dir` function.

In [43]:

```
dir(iDEA.methods)
```

Out[43]:

```
['Enum',
 'Methods',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'hartree',
 'hartree_fock',
 'hybrid',
 'interacting',
 'lda',
 'non_interacting']
```

We can investigate a given method in a little more detail:

In [44]:
```
help(iDEA.methods.hartree_fock)
```

```
Help on module iDEA.methods.hartree_fock in iDEA.methods:

NAME
    iDEA.methods.hartree_fock - Contains all Hartree Fock functionality and s
olvers.

FUNCTIONS
    exchange_potential_operator(s: iDEA.system.System, p: numpy.ndarray) -> n
umpy.ndarray
        Compute the exchange potential operator.

        | Args:
        |     s: iDEA.system.System, System object.
        |     p: np.ndarray, Charge density matrix.

        | Returns:
        |     Vx: np.ndarray, Exchange potential energy operator.

    hamiltonian(s: iDEA.system.System, up_n: numpy.ndarray, down_n: numpy.nda
rray, up_p: numpy.ndarray, down_p: numpy.ndarray, K: numpy.ndarray = None, Ve
xt: numpy.ndarray = None) -> numpy.ndarray
        Compute the Hamiltonian from the kinetic and potential terms.

        | Args:
        |     s: iDEA.system.System, System object.
        |     up_n: np.ndarray, Charge density of up electrons.
        |     down_n: np.ndarray, Charge density of down electrons.
        |     up_p: np.ndarray, Charge density matrix of up electrons.
        |     down_p: np.ndarray, Charge density matrix of down electrons.
        |     K: np.ndarray, Single-particle kinetic energy operator [If None
this will be computed from s]. (default = None)
        |     Vext: np.ndarray, Potential energy operator [If None this will
be computed from s]. (default = None)

        | Returns:
        |     H: np.ndarray, Hamiltonian, up Hamiltonian, down Hamiltonian.

    propagate(s: iDEA.system.System, state: iDEA.state.SingleBodyState, v_ptr
b: numpy.ndarray, t: numpy.ndarray, hamiltonian_function: collections.abc.Cal
lable = None, restricted: bool = False) -> iDEA.state.SingleBodyEvolution
        Propagate a set of orbitals forward in time due to a dynamic local pe
rtubation.
```

```
        | Args:
        |     s: iDEA.system.System, System object.
        |     state: iDEA.state.SingleBodyState, State to be propigated.
        |     v_ptrb: np.ndarray, Local perturbing potential on the grid of t
and x values, indexed as v_ptrb[time,space].
        |     t: np.ndarray, Grid of time values.
        |     hamiltonian_function: Callable, Hamiltonian function [If None t
his will be the non_interacting function]. (default = None)
        |     restricted: bool, Is the calculation restricted (r) on unrestri
cted (u). (default=False)

        | Returns:
        |     evolution: iDEA.state.SingleBodyEvolution, Solved time-dependen
t evolution.

    solve(s: iDEA.system.System, k: int = 0, restricted: bool = False, mixin
g: float = 0.5, tol: float = 1e-10, initial: tuple = None, silent: bool = Fal
se) -> iDEA.state.SingleBodyState
        Solves the Schrodinger equation for the given system.

        | Args:
        |     s: iDEA.system.System, System object.
        |     k: int, Energy state to solve for. (default = 0, the ground-sta
te)
        |     restricted: bool, Is the calculation restricted (r) on unrestri
cted (u). (default=False)
        |     mixing: float, Mixing parameter. (default = 0.5)
        |     tol: float, Tollerance of convergence. (default = 1e-10)
        |     initial: tuple. Tuple of initial values used to begin the self-
consistency (n, up_n, down_n, p, up_p, down_p). (default = None)
        |     silent: bool, Set to true to prevent printing. (default = Fals
e)

        | Returns:
        |     state: iDEA.state.SingleBodyState, Solved state.

    total_energy(s: iDEA.system.System, state: iDEA.state.SingleBodyState) ->
float
        Compute the total energy.

        | Args:
        |     s: iDEA.system.System, System object.
        |     state: iDEA.state.SingleBodyState, State. (default = None)

        | Returns:
        |     E: float, Total energy.

DATA
    name = 'hartree_fock'

FILE
    /home/jack/projects/iDEA/.venv/lib/python3.8/site-packages/iDEA_latest-0.
1.6-py3.8.egg/iDEA/methods/hartree_fock.py
```

We can solve our system using **LDA** and compare this to our exact result:

In [45]:

```
ground_state_lda = iDEA.methods.lda.solve(s, k=0)
```

```
iDEA.methods.lda.solve: convergence = 6.9333e-11, tolerance = 1e-10
```

In [46]:

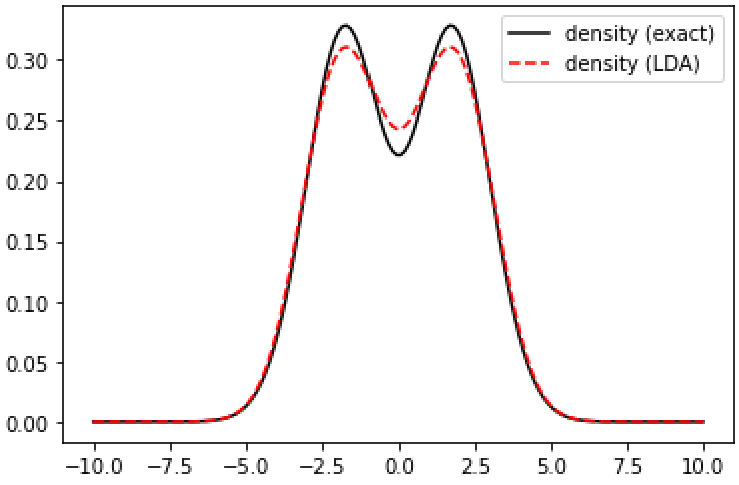
```
n_exact = iDEA.observables.density(s, state=ground_state)
```

In [47]:

```
n_lda = iDEA.observables.density(s, state=ground_state_lda)
```

In [48]:

```
plt.plot(s.x, n_exact, "k-", label="density (exact)")
plt.plot(s.x, n_lda, "r--", label="density (LDA)")
plt.legend()
plt.show()
```

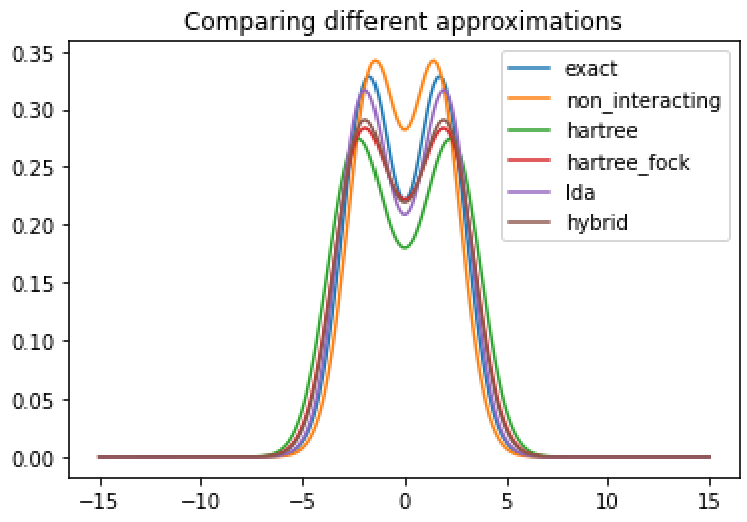


`iDEA` makes it easy to compare many different approximations. For example, **we can compare the first-excited state charge density, for the two electron (ud) QHO for all of the methods using the following simple code:**

In [49]:

```
x = np.linspace(-15, 15, 200)
v_ext = 0.5 * 0.25**2 * x**2
v_int = iDEA.interactions.softened_interaction(x)
s = iDEA.system.System(x, v_ext, v_int, electrons='ud')


state_exact = iDEA.methods.interacting.solve(s, k=1)
n_exact = iDEA.observables.density(s, state_exact)
plt.plot(s.x, n_exact, label="exact")

for method in iDEA.iterate_sb_methods:
    state = method.solve(s, k=1, tol=1e-6)
    n = iDEA.observables.density(s, state)
    plt.plot(s.x, n, label=method.name)
plt.title("Comparing different approximations")
plt.legend()
plt.show()
```

```
iDEA.methods.non_interacting.solve: convergence = 9.021e-12, tolerance = 1e-0
6
iDEA.methods.hartree.solve: convergence = 7.6081e-07, tolerance = 1e-06
iDEA.methods.hartree_fock.solve: convergence = 7.9506e-07, tolerance = 1e-06
iDEA.methods.lda.solve: convergence = 9.5944e-07, tolerance = 1e-06
iDEA.methods.hybrid.solve: convergence = 8.8926e-07, tolerance = 1e-06
```



7. Reverse Engineering.

`iDEA` also gives you the ability to reverse-engineer charge densities. **This tells you what fictitious system would need to be solved using a given method, to reproduce a target density.**

The reverse function provides this functionality:

In [50]:
```
help(iDEA.reverse_engineering.reverse)
```

```
Help on function reverse in module iDEA.reverse_engineering:

reverse(s: iDEA.system.System, target_n: numpy.ndarray, method: collections.a
bc.Container, v_guess: numpy.ndarray = None, mu: float = 1.0, pe: float = 0.
1, tol: float = 1e-12, silent: bool = False, **kwargs) -> iDEA.state.State
    Determines what ficticious system is needed for a given method, when solv
ing the system, to produce a given target density.
    If the given target density is from solving the interacting electron prob
lem (iDEA.methods.interacting), and the method is the non-interacting electro
n solver (iDEA.methods.non_interacting)
    the output is the Kohn-Sham system.

    The iterative method used is defined by the following formula:
    .. math:: \mathrm{V}_\mathrm{ext} \rightarrow \mu * (\mathrm{n}^p - \math
rm{target_n}^p)

    | Args:
    |     s: iDEA.system.System, System object.
    |     target_n: np.ndarray, Target density to reverse engineer.
    |     method: Container, The method used to solve the system.
    |     v_guess: np.ndarray, The initial guess of the fictitious potential.
(default = None)
    |     mu: float = 1.0, Reverse engineering parameter mu. (default = 1.0)
    |     pe: float = 0.1, Reverse engineering parameter p. (default = 0.1)
    |     tol: float, Tollerance of convergence. (default = 1e-12)
    |     silent: bool, Set to true to prevent printing. (default = False)
    |     kwargs: Other arguments that will be given to the method's solve fu
nction.

    | Returns:
```

```
|     s_fictitious: iDEA.system.System, fictitious system object.
```

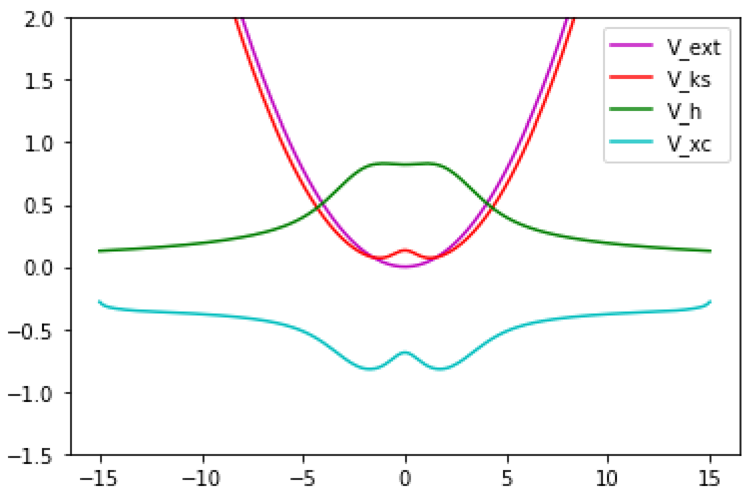
This is a general method, but can be used to perform many common tasks in condenced matter physics.

For example, we can determine the exact Kohn-Sham (KS) and exchange-correlation (xc) potential of a system, by asking: **What fictitious system, when solved with the non-interacting method, gives us the same density as that solved by the interacting method.** The external potential of this fictitious system is none other than the exact KS potential, from which the exact xc potential can be found:

In [51]:

```
x = np.linspace(-15, 15, 200)
v_ext = 0.5 * 0.25**2 * x**2
v_int = iDEA.interactions.softened_interaction(x)
s = iDEA.system.System(x, v_ext, v_int, electrons='uu')


state = iDEA.methods.interacting.solve(s, k=0)
n = iDEA.observables.density(s, state)


s_fictitious = iDEA.reverse_engineering.reverse(s, target_n=n, method=iDEA.me
state_fictitious = iDEA.methods.non_interacting.solve(s_fictitious, k=0)


v_ks = s_fictitious.v_ext
n_fictitious = iDEA.observables.density(s, state_fictitious)
v_h = iDEA.observables.hartree_potential(s_fictitious, n_fictitious)
v_xc = v_ks - s.v_ext - v_h


plt.plot(s_fictitious.x, s.v_ext, 'm-', label="V_ext")
plt.plot(s_fictitious.x, v_ks, 'r-', label="V_ks")
plt.plot(s_fictitious.x, v_h, 'g-', label="V_h")
plt.plot(s_fictitious.x, v_xc, 'c-', label="V_xc")
plt.legend()
plt.ylim([-1.5, 2.0])
plt.show()
```

```
iDEA.reverse_engineering.reverse: convergence = 1.0131e-12, tolerance = 1e-12
iDEA.methods.non_interacting.solve: convergence = 6.8384e-12, tolerance = 1e-
10
```



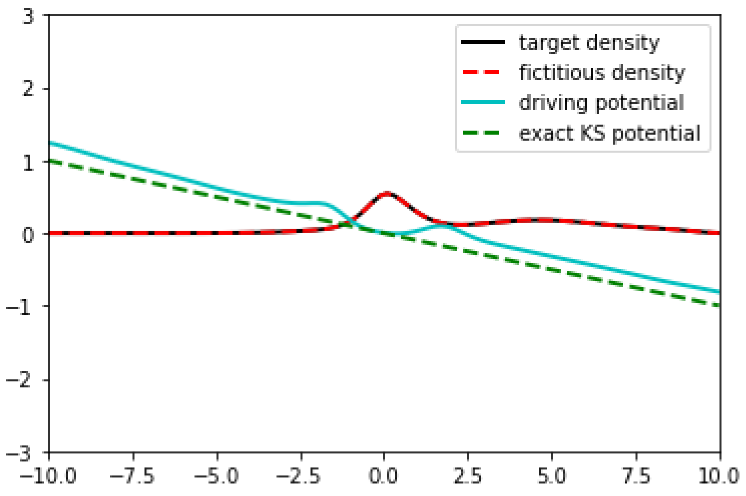
Reverse-engineering can also be used to determine what time-dependent potential is needed in a fictitious evolution, in order to recover a given target density. A full example of this is shown below:

In [52]:
```
import scipy.special as spspec

x = np.linspace(-10, 10, 100)
v_ext = -2.0 / (abs(x) + 1.0)
v_int = iDEA.interactions.softened_interaction(x)
electrons = 'uu'
s = iDEA.system.System(x, v_ext, v_int, electrons)


t = np.linspace(0, 10, 100)
v_ptrb = np.zeros(shape=t.shape+x.shape)
for j, ti in enumerate(t):
    v_ptrb[j,:] = -0.1 * x * 0.5*(spspec.erf(ti - 2.0) + 1.0)


state = iDEA.methods.interacting.solve(s, k=0)
target_n = iDEA.observables.density(s, state=state)
s_fictitious = iDEA.reverse_engineering.reverse(s, target_n, method=iDEA.meth
state_fictitious = iDEA.methods.non_interacting.solve(s_fictitious)


evolution = iDEA.methods.interacting.propagate(s, state, v_ptrb, t)
target_n = iDEA.observables.density(s, evolution=evolution)
evolution_fictitious, error = iDEA.reverse_engineering.reverse_propagation(s_
evolution_fictitious = iDEA.methods.non_interacting.propagate(s_fictitious, s


for ti in range( len(t) ):
    evolution_fictitious.v_ptrb[ti,:] -= evolution_fictitious.v_ptrb[ti,int(0


n = iDEA.observables.density(s, evolution=evolution_fictitious)
v_ext = iDEA.observables.external_potential(s)
v_h = iDEA.observables.hartree_potential(s, n)


fig = plt.figure()
ax = plt.axes(xlim=(s.x[0], s.x[-1]), ylim=(-3.0, 3.0))
line1, = ax.plot([], [], lw=2, c="k", label="target density")
line2, = ax.plot([], [], lw=2, c="r", ls="--", label="fictitious density")
line3, = ax.plot([], [], lw=2, c="c", label="driving potential")
line4, = ax.plot([], [], lw=2, c="g", ls="--", label="exact KS potential")
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    line3.set_data([], [])
    line4.set_data([], [])
    return line1, line2, line3, line4
def animate(i):
    line1.set_data(s.x, n[i,:])
    line2.set_data(s.x, target_n[i,:])
    line3.set_data(s.x, evolution_fictitious.v_ptrb[i,:])
    line4.set_data(s.x, v_ptrb[i,:])
    return line1, line2, line3, line4
plt.legend()
anim = FuncAnimation(fig, animate, init_func=init, frames=100, interval=10, b
HTML(anim.to_html5_video())
```

```
iDEA.reverse_engineering.reverse: convergence = 1.1004e-12, tolerance = 1e-12
iDEA.methods.non_interacting.solve: convergence = 2.7785e-11, tolerance = 1e-
10
iDEA.methods.interacting.propagate:
iDEA.reverse_engineering.reverse_pro
iDEA.methods.non_interacting.propaga
```

Out[52]:

0:00 / 0:01



8. A full example: Investigating 1D hydrogen molecule.

Now you know all the basics of using iDEA to investigate and simulate quantum systems, here we will show you a "real-world" example.

In the following section of code, we will compute the exact bonding curve of the 1D hyrogen molecule, and compare it to a conventional figure.

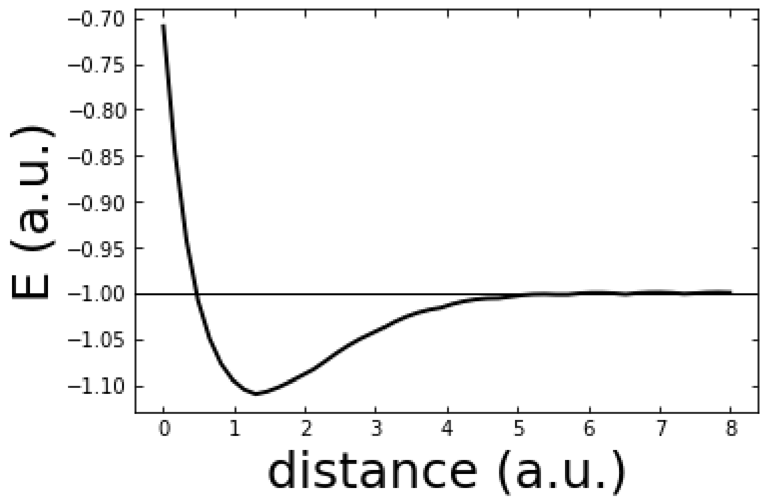
In [53]:

```python
# Build the base system.
x = np.linspace(-10, 10, 200)
v_ext = np.zeros_like(x)
v_int = iDEA.interactions.softened_interaction(x)
electrons = 'ud'
s = iDEA.system.System(x, v_ext, v_int, electrons)


# Function to build the hydrogen molecule at each given distance d.
d = 8.0
def v(x):
    return -1.0/(np.abs(x + 0.5*d) + 1.0) -1.0/(np.abs(x - 0.5*d) + 1.0)
```

```
# Define range of bond distances.
distances = np.linspace(0.0, 8.0, 50)
energies = np.zeros_like(distances)


# Compute exact electron energy at each distance.
for i in range(distances.shape[0]):
    d = distances[i]
    print('distance = {0}'.format(d))
    s.v_ext = np.copy(v(x))
    state = iDEA.methods.interacting.solve(s)
    energies[i] = state.energy


# Add the proton energy.
E_proton = 1.0 / (np.abs(distances) + 1.0)
energies += E_proton


# Plot energy curve.
plt.axhline(y=-1.0, xmin=0.0, xmax=8.2, linewidth=1, color='k')
plt.plot(distances, energies, linestyle='-', color='k', linewidth=2)
plt.xlabel('distance (a.u.)', size=25)
plt.ylabel('E (a.u.)', size=25)
plt.tick_params(top='on', right='on')
plt.tick_params(direction='in')
plt.gcf().subplots_adjust(left=0.18)
plt.gcf().subplots_adjust(bottom=0.18)
plt.plot()
```

```
distance = 0.0
distance = 0.16326530612244897
distance = 0.32653061224489793
distance = 0.4897959183673469
distance = 0.6530612244897959
distance = 0.8163265306122448
distance = 0.9795918367346939
distance = 1.1428571428571428
distance = 1.3061224489795917
distance = 1.4693877551020407
distance = 1.6326530612244896
distance = 1.7959183673469385
distance = 1.9591836734693877
distance = 2.1224489795918364
distance = 2.2857142857142856
distance = 2.4489795918367343
distance = 2.6122448979591835
distance = 2.7755102040816326
distance = 2.9387755102040813
distance = 3.1020408163265305
distance = 3.265306122448979
distance = 3.4285714285714284
distance = 3.591836734693877
distance = 3.7551020408163263
distance = 3.9183673469387754
distance = 4.081632653061225
distance = 4.244897959183673
distance = 4.408163265306122
distance = 4.571428571428571
distance = 4.73469387755102
distance = 4.897959183673469
distance = 5.061224489795918
distance = 5.224489795918367
distance = 5.387755102040816
distance = 5.551020408163265
distance = 5.7142857142857135
distance = 5.877551020408163
distance = 6.040816326530612
distance = 6.204081632653061
distance = 6.367346938775509
distance = 6.530612244897958
distance = 6.693877551020408
distance = 6.857142857142857
distance = 7.020408163265306
distance = 7.183673469387754
distance = 7.346938775510203
distance = 7.5102040816326525
distance = 7.673469387755102
distance = 7.836734693877551
distance = 8.0
```

```
Out[53]: []
```



We can see this follows the expected curve:

Curve

Now we recommend, using the tools we have walked through in this tutorial, you gain more experience with the iDEA code by trying to solve some of these more advanced open-ended problems:

1. What does the bonding curve from part 8 look like if the electrons have the same spin?
2. What is the exact xc potential for the three electron 1D lithim atom.
3. Which approximation yeilds the most accurate density matix for the QHO.
4. What kind of field is needed to ionise an electron out of an atom in real time?
5. Can you modify the occupations of a Hartree-Fock calculation, to yeild the exact density?