iDEA Documentation

Release 0.1.0

Sean Adamson Jacob Chapman Thomas Durrant
Razak Elmaslmane Mike Entwistle Rex Godby
Matt Hodgson Piers Lillystone Aaron Long
Robbie Oliver James Ramsden Ewan Richardson
Matthew Smith Leopold Talirz Jack Wetherell

Dec 15, 2018

Contents

CHAPTER 1

Using iDEA

1.1 Getting iDEA

1.1.1 Installation requirements

- Python 3.3 or later
- numpy 1.10 or later
- scipy 0.17 or later
- Cython 0.22 or later
- *(optional)* matplotlib 1.4 or later for post-processing

1.1.2 Installing iDEA

```
# Clone from the central repository
git clone https://github.com/godby-group/idea-public.git idea-public

# Install & compile iDEA for your unix user
# (including packages for generating the documentation)
cd idea-public
pip install --user -e .[doc]

# Run example calculation
python run.py
```

1.1.3 Updating iDEA

```
# Pull all changes from central git repository
git pull origin master

# Remove the compiled cython modules
python setup.py clean --all

# Recompile the cython modules
python setup.py build_ext --inplace
```

1.1.4 Generating the documentation

A recent version of the documentation can be found on the iDEA web page. If you are making changes to the code and/or the documentation, you may need to generate the documentation by yourself:

```
cd doc
bash make_doc.sh
# find html documentation in _build/html/index.html
# find test coverage report in _build/coverage/index.html
```

Besides HTML, the iDEA documentation can also be compiled as a pdf. If you have a LaTeX distribution installed on your system, simply do:

```
cd doc
make latexpdf
```

1.2 Running iDEA

As it is a python package there are many different ways of running iDEA.

1.2.1 Using the iDEA code directly

Simply edit the parameters file `parameters.py` and run

```
python run.py
```

In order not to overwrite results from different calculations, make sure to choose different run names for different inputs

```
### Library imports
from __future__ import division
from iDEA.input import InputSection, SystemSection
import numpy as np


### Run parameters
run = InputSection()
run.name = 'run_name'                  #: Name to identify run. Note: Do not use spaces␣
↪or any special characters (.~[]{}<>?/\)
run.time_dependence = False            #: Run time-dependent calculation
run.verbosity = 'default'              #: Output verbosity ('low', 'default', 'high')
run.save = True                        #: Save results to disk when they are generated
```

(continues on next page)

(continued from previous page)

```
run.module = 'iDEA'                   #: Specify alternative folder (in this␣
↪directory) containing modified iDEA module
run.NON = True                        #: Run Non-Interacting approximation
run.LDA = False                       #: Run LDA approximation
run.HF = False                        #: Run Hartree-Fock approximation
run.HYB = False                       #: Run Hybrid (HF-LDA) calculation
run.EXT = True                        #: Run Exact Many-Body calculation
```

1.2.2 Using the iDEA package in a python script

Since iDEA is designed as a python package, it can be run from everywhere, if you let your python installation know where the package is located. During the installation of iDEA the *idea-public* directory should have been added to PYTHONPATH. To test this has worked simply perform the following

```
cd $test_folder                   # some folder you have created
cp $path_to_iDEA/parameters.py . # where you have downloaded iDEA
cp $path_to_iDEA/run.py .
python run.py
```

Here, we are running iDEA much in the same way as before but your `$test_folder` can be located anywhere on your computer.

The main advantage of having an iDEA python package is that you can access its functionality directly in a python script.

The example below uses a python loop to converge the grid spacing for an iDEA calculation of a test system of non-interacting electrons in a harmonic well.

```
from iDEA.input import Input

# read parameters file
inp = Input.from_python_file('parameters.py')
inp.run.verbosity = 'low'

# Converging xmax parameter
for xmax in [4,6,8,10]:
    # Note: the dependent sys.deltax is automatically updated
    inp.sys.xmax = xmax

    # perform checks on input parameters
    inp.check()
    inp.execute()
    E = inp.results.non.gs_non_E
    print(" xmax = {:4.1f}, E = {:6.4f} Ha".format(xmax,E))
```

In order to run this example, do

```
cd $path_to_iDEA/examples/06_convergence
python run.py  # assuming you already added iDEA to your PYTHONPATH
```

1.2.3 Using the iDEA package in an ipython shell

As iDEA is a python package it can also be run from the interactive python shell (`ipython`). This is particularly useful as ipython has a convenient auto-complete feature. The following example can be run in an ipython shell line-by-line.

```
from iDEA.input import Input

# import parameters file
pm = Input.from_python_file('parameters.py')

# change parameters
pm.run.EXT = True
pm.run.NON = True
print(pm.run)

# run jobs
results = pm.execute()

# plot the relevant results
x = pm.space.grid
plt.plot(x, results.ext.gs_ext_den, label='exact')
plt.plot(x, results.non.gs_non_den, label='non-interacting')
plt.legend()
plt.show()
```

1.3 Running ViDEO

ViDEO is a python script used to process the raw output files generared by iDEA into easy to use .dat data files, .pdf plots and .mp4 animations

1.3.1 Using ViDEO directly

Simply naviage to the relevent outputs folder

```
cd outputs/run_name/
```

and run the script

```
python ViDEO.py
```

You will be promted for information to determine the result you want to process, and what files should be generated. ViDEO makes use of the file convention of outpus used in iDEA, as all pickle files are names according to the following convention

```
{gs/td}_{appoximation}_{quantity}.db
```

for example, the ground-state non-interacting density

```
gs_non_den.db
```

1.3.2 Using ViDEO on results generated from a script

Simply naviage to the relevent outputs folder (assuming `run.save = True` in the relevent input file)

```
cd outputs/run_name/
```

and run the command

```
ViDEO.py
```

CHAPTER 2

Examples

You'll find a collection of jupyter notebooks in the `examples/` subdirectory. Once you've installed iDEA on your computer, you can run these examples interactively via

```
cd examples/
jupyter notebook
```

If you prefer to get a first impression without installing iDEA, you find static HTML versions of the examples below.

2.1 Getting started: an interactive tutorial for using iDEA

2.1.1 Guide to using jupyter notebooks

Jupyter notebooks allow code to be edited and run within the notebook using an iPython shell, giving ouputs and error messages. This allows the user to play around with the code as much as they'd like. Much like the iPython shell, one closed and reopened, all the relevent code boxes must be executed once again (everything will once again be default). Left of the code box shows the order of execution. If the code is currently executing, it will display an asterisk. **Press 'Shift + Enter' to execute the code**.

When exiting the Jupyter notebook, make sure to save the click 'File' then 'Close and Halt' to stop the notebook running. Note that, like a Python shell, all changes made to source code and data created will be lost (apart from whatever the last output was when the code block was ran).

2.1.2 The iDEA code

The iDEA code can solve the time-independent as well as the time-dependent Schrödinger equation for 2-3 electrons in one-dimensional real space. It can then be compared with known methods for solving many-body problems, such as various flavours of density functional theory and many-body perturbation theory.

Compared to lattice models, such as the Anderson impurity model, this approach allows to treat exchange and correlation throughout the system and provides additional flexibility in bridging the gap between model systems and ab initio descriptions.

For a list of features, see the iDEA homepage.

2.1.3 1. Running iDEA

In the following, we assume that you have installed idea following the installation instructions on the web site.

Setting up

All iDEA files are found in `iDEAL` so this needs to be made the current directory.

- `parameters.py` contains all the parameters for iDEA and editing the varaibles within this will change what system iDEA will solve for
- `iDEA/input.py` in the iDEA folder which will allow 'parameters.py' to be used. Hence:

[2]:
```
from iDEA.input import Input
pm = Input.from_python_file('parameters.py')
```

Parameters

The parameters file decides exactly what simulation is run. The main parameters are the `run` and `system` parameters. Printing them to the screen, it can be seen what their default setting is. Do not worry about what they all mean for the moment as they will be described, in detail, later on.

[3]:
```
print(pm.run)
print(pm.sys)
```

```
name = 'run_name'
time_dependence = False
verbosity = 'default'
save = True
module = 'iDEA'
NON = True
LDA = False
MLP = False
HF = False
EXT = True
MBPT = False
HYB = False
LAN = False
MET = False

NE = 2
grid = 201
stencil = 3
xmax = 10.0
tmax = 1.0
imax = 1001
acon = 1.0
interaction_strength = 1.0
im = 0
v_ext = <function v_ext at 0x7f0518dea7b8>
v_pert = <function v_pert at 0x7f0518dea9d8>
v_pert_im = <function v_pert_im at 0x7f0518deaa60>
```

As can be seen above, the run parameters decide what method is used. It should be noted that multiple methods can be set to true and the iDEA code will do both in a single run. The system parameters can alter things like the number of electrons, grid size and maximum time. The potentials cannot be shown, as they are functions. The parameter file needs to be edited directly to change the function.

It is EXTREMELY important to set a run name before you start the simulation. This creates a subdirectory in which all the results are contained. If you do not set a run name, it will default to 'run_name' and before long, it will be hard to keep track of the results and will overwrite previous results of the same time. This is obviously not ideal.

The current setting is solving the time-independent Schrodinger equation for two electrons exactly, finding the ground-state many-electron wavefunction (by propagation in imaginary time) and using DFT with the non-interaction approximation.

The individual parameter values can be overwritten in iPython, and hence this notebook, like so:

```
[4]: pm.run.LDA = False
     pm.sys.NE = 2
     # Create a new subdirectory to contain the results onwards from this. Note
     # once the kernal is restarted, all this setup needs to be repeated.
     pm.run.name = 'get_started_basics'
```

The iDEA code can now be executed. The object 'results' recieves the code output so that the results can be referenced in matplotlib for plotting.

Note that, depending on the complexity of the simulation and the number of jobs set to complete, the results may take some time to generate (note the asterisk). For the default setup, it takes approximately 10 seconds to complete but do note that something like a time-dependent simulation propagating for a reasonable time may take hours.

Now to execute the code:

```
[5]: pm.check()  # perform a few sanity checks
     results = pm.execute()
```

```

                *    ****     *****       *
                     *   *    *          * *
                *    *    *   *         *   *
                *    *     *  *****    *     *
                *    *    *   *       *********
                *    *   *    *      *         *
                *    ****     ***** *           *

  +---------------------------------------+
  |          Interacting Dynamic Electrons Approach          |
  |              to Many-Body Quantum Mechanics              |
  |                                                          |
  |                      Release 2.3.0                       |
  |                     git commit hash                      |
  |         8fafd8504e042d86da5950e9fc2fa0b2497244ff         |
  |                                                          |
  |        Created by Jacob Chapman, Thomas Durrant,         |
  |       Razak Elmaslmane, Mike Entwistle, Rex Godby,       |
  |       Matt Hodgson, Piers Lillystone, Aaron Long,        |
  |            Robbie Oliver, James Ramsden, Ewan            |
  |      Richardson, Matthew Smith, Leopold Talirz and       |
  |                      Jack Wetherell                      |
  |                                                          |
  |                    University of York                    |
```

(continues on next page)

(continued from previous page)

```
  +----------------------------------------+

run name: get_started_basics
NON: constructing arrays
NON: computing ground state density
NON: ground-state energy = 0.49988
EXT: constructing arrays
EXT: imaginary time propagation
EXT: t = 96.92000, convergence = 9.955330352475543e-13K
EXT: ground-state converged
EXT: ground-state energy = 0.75310
all jobs done
```

All the data from the simulation will be placed in the outputs directory, specifically: **outputs/run_name/raw** where run_name is the default subdirectory created if the used hasn't changed the

2.1.4 2. Plotting the results

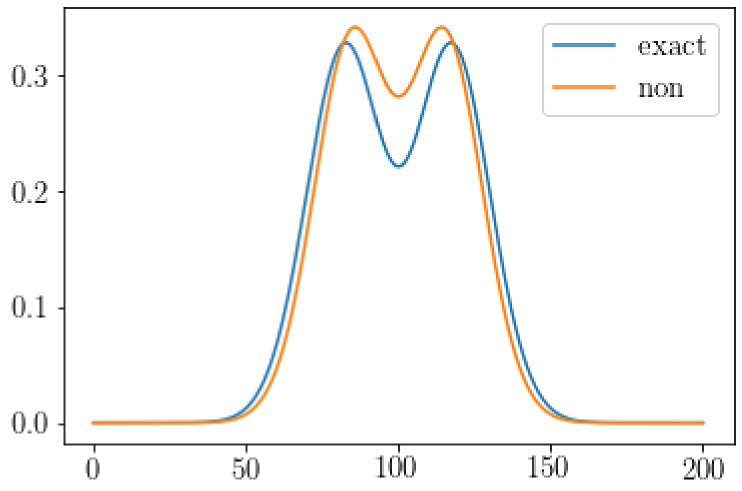
Matplotlib integrates well with python so this is recommended. As usual, this needs to be imported:

```
[6]: import matplotlib.pyplot as plt
```

The electron density for the exact solution and DFT with the non-interaction will be plotted. The results are organised by **result.resultType.gs_resultType_info** where in this case, **resultType** is 'ext' (exact) and 'non' (non-interaction) and **info** is 'den' (electron density). *Colour-reference*

```
[7]: plt.plot(results.ext.gs_ext_den, label='exact')
     plt.plot(results.non.gs_non_den, label='non')
     plt.legend()
```

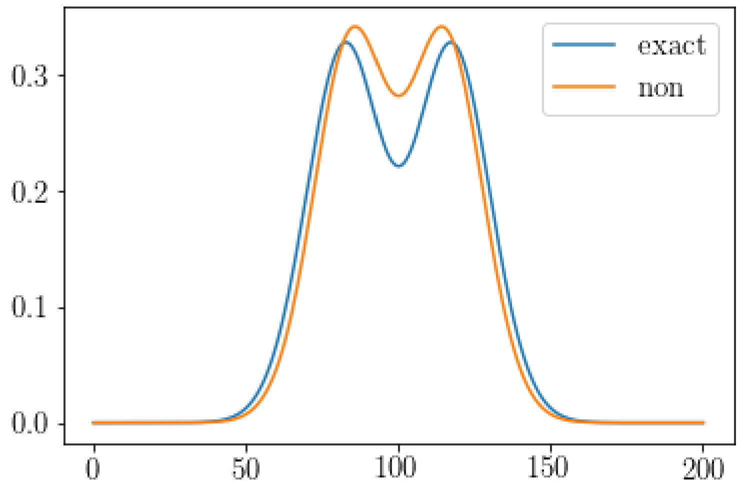
```
[7]: <matplotlib.legend.Legend at 0x7f04ec087fd0>
```



```
[8]: plt.show()
```

Although this is a perfectly acceptable way of plotting results, there is another way, which is in keeping more with the object-oriented approach that has, thus far and in the future, been taken. The results need to be imported as such:

```
[9]: from iDEA.results import Results as rs
     # Make sure the run name is correct first
     pm.run.name = 'get_started_basics'
     # The data is imported as follows:
     example_plot_ext = rs.read('gs_ext_den',pm)
     example_plot_non = rs.read('gs_non_den',pm)

     # Note that a variable needs to be created containing the data. The string,
     # which is the name of the data file does not contain the file extension.
     # Note that pm is also an argument.

     # Next, plotted as follows:
     plt.plot(example_plot_ext,label='exact')
     plt.plot(example_plot_non,label='non')
     plt.legend()
     plt.show()
```



2.1.5 3. Changing the parameters

Be prepared; there are many parameters that, at first, might seem confusing (most of the specialist parameters are explained in section 6). However, there is no need to worry, since iDEA contains a lot of specialised functionalities that can be ignored. To emphasise this, the core functionalities will be in bold and the more specialised, not. For the sake of completeness, however, there will be a brief description of all parameters. Again not that it will not be necessary to understand these to be able to use iDEA effectively for most purposes.

As a reminder, to find the parameters, when coding type the relevent parameter (run, sys etc.) followed by a full stop, then press 'Tab' to autocomplete.

As mentioned before, the run parameters decide what is ran, where multiple things can be run at once. These take either the value True or False: * **run.time_dependence** - Time-dependence * **run.NON** - The Non-interaction approximation * **Density Functional Theory** * **run.LDA** - Local Density Approximation * **run.HF** - Hartree-Fock approximation

Each of these have further parameters which can be adjusted, which will be discussed later.

There are also system parameters, which change more general aspects of the simulation: * **sys.NE** - Number of electrons *(2/3)* * **sys.grid** - Number of grid points (odd to include the zero) * sys.stencil - Descretisation of 2nd derivative (3,5 or 7) {higher means more accuracy, more memory usage} * **sys.xmax** - Size of the system * **sys.tmax** - Total real time * **sys.imax** - Number of real-time iterations * sys.acon - Smoothing of Coulomb interaction * sys.interaction_strength - Scales strength of Coulomb interaction * sys.im - Using imaginary potentials (0=no/1=yes)

The number of grid points determines the accuracy of how the discrete system simulations the true continuous system. 201 is a good default.

Imaginary potentials are introduced at the boundary. They are used to create more accurate results for systems where electrons are colliding off one another.

2.1.6 4. Potential functions

Note that we are working in Hatree atomic units ($\hbar = e = m = 4\pi\epsilon_0 = 1$), which means the unit of length is the Bohr radius (53 pm) and the unit of energy is the Hartree (27.2 eV).

The potential functions, such as the external, T-D pertubation and imaginary pertubation potentials can be definined in a function, and set to the parameters as follows:

```
[10]: def v_ext(x):
          'Initial external potential'
          return 0.5*(0.25**2)*(x**2)
      pm.sys.v_ext = v_ext
```

Any reasonable potential can be defined this way.

2.1.7 5. Time Dependence

The time-dependent solutions can be found for exact and non-interacting. Let us now set up the parameters in mind for animating the results; extending the imaginary time propagation, number of grid points and total time. Bear in mind that, so the accuracy is not affected, if the imaginary time is increased, the number of grid points need to be scaled with it.

```
[11]: pm.run.time_dependence = True
      pm.sys.grid= 301
      pm.sys.imax = 15001
      pm.sys.tmax = 15
      pm.check()
      #excecution commented out as T-D takes hours. Uncomment if you want to generate new␣
      ↪results
      #results=pm.execute()
```

Animating the results

The time-dependent simulation will produce results, such as the electron density for each timestep. Therefore, the results may be animated so that the results can be interpreted in a meaningful way.

This can again be done through matplotlib.

```
[12]: import numpy as np
      import matplotlib.pyplot as plt
      from matplotlib import animation, rc
      from IPython.display import HTML
      from iDEA.input import Input
```

Since the time-dependent run takes a long time, it is useful to read the saved results from the output file so that the iDEA code does not have to be run every time we want to create an animation.

[16]:
```
from iDEA.results import Results as rs
file_name = 'td_ext_den'  #Change this to the ".db" file name you want to see, not␣
↪including the .db part
array = rs.read(file_name, pm)
print(np.shape(array))
```

```
(15001, 301)
```

Here, the first index corresponds to the time grid and the second to the spatial grid.

The background now needs to be created for the animation. The x-axis corresponds to the grid and the y axis the density, in this case.

[17]:
```
fig, ax = plt.subplots()
ax.set_xlim((0, pm.sys.grid))
ax.set_ylim((0, 0.6))
line, = ax.plot([], [], lw=2)

def init():
    line.set_data([], [])
    return (line,)

def animate(i):
    x = np.arange(0,pm.sys.grid)
    y = array[10*i,x]             # 10*i so that every 10th frame is animated
    line.set_data(x, y)
    return (line,)

rc('animation', html='html5')

animation.FuncAnimation(fig, animate, init_func=init,save_count =int(pm.sys.imax*0.1),

                               interval=10, blit=True)
```

[17]:
```
<matplotlib.animation.FuncAnimation at 0x7fde9cb64f98>
```

Freestyle

Now that all of the basics have been covered, use the cell below to change the parameters to test your understanding.

[ ]:
```
# Use this cell to test understanding of what was covered so far
```

Once you are comfortable with all that has been discussed thus far, the next part of the tutorial will give a detailed description of all the parameters that can be changed. Each approximation/calculation has a further set of parameters which can be modified. There will be exercises to complete to test understanding, with hints at the bottom of the page if you encounter difficulty.

2.1.8 6. Further Parameters

Exact Many-body Calculation

The exact many-body calculation is done by propagating the wavefunction in imaginary time, which eventually converges on the ground-state energy. The exact (ext) parameters are as follows:

- **ext.itol** - Tolerance of imaginary time propagation (~ 1e-12)

- **ext.itol_solver** - Tolerance of linear solver in imaginary time propagation (~ 1e-14)
- **ext.rtol_solver** - Tolerance of linear solver in real time propagation (~ 1e-12)
- **ext.itmax** - Total imaginary time
- **ext.iimax** - Imaginary time iterations
- **ext.ideltat** - Imaginary time step (DERIVED)
- **ext.RE** - Reverse engineer many-body density
- **ext.OPT** - Calculate the external potential for the exact density
- **ext.excited_states** - Number of excited states to calculate (0 for only ground-state)
- **ext.elf_gs** - Calculate ELF for the ground-state of the system
- **ext.elf_es** - Calculate ELF for the excited-states of the system
- **ext.elf_td** - Calculate ELF for the time-dependent part of the system
- **ext.psi_gs** - Save the reduced ground-state wavefunction to file
- **ext.psi_es** - Save the reduced excited-state wavefunctions to file
- **ext.initial_psi** - initial wavefunction: qho, non, hf, lda1/2/3/heg, ext or wavefunction from previous run (e.g run_name)

Tolerance describes the difference between, say for example, densities between iterations before a successful convergence has occured.

ELF is the Electron Localisation Function which describes how localised the electrons are. This can be calculated for the ground-state, excited states and time-dependent system.

Exercise 1

Perfom exact many-body calculations for the first excited state and calculate the relevent ELF.

[ ]:
```
pm.run.LDA = True
pm.sys.NE = 3
pm.run.verbosity = 'default'
pm.lda.NE = 'heg'
print(pm.run)
results = pm.execute()
```

```
name = 'run_name'
time_dependence = True
verbosity = 'default'
save = True
module = 'iDEA'
NON = True
LDA = True
MLP = False
HF = False
EXT = True
MBPT = False
LAN = False



                *    ****     *****       *
                     *   *    *          * *
```

(continues on next page)

(continued from previous page)

```
                *    *    *   *         *   *
                *    *     *  *****    *     *
                *    *    *   *       *********
                *    *   *    *      *         *
                *    ****     ***** *           *

  +---------------------------------------+
  |          Interacting Dynamic Electrons Approach          |
  |              to Many-Body Quantum Mechanics              |
  |                                                          |
  |                      Release 2.1.0                       |
  |                                                          |
  |       Created by Piers Lillystone, James Ramsden,        |
  |       Matt Hodgson, Thomas Durrant, Jacob Chapman,       |
  |         Jack Wetherell, Mike Entwistle, Matthew          |
  |           Smith, Aaron Long and Leopold Talirz           |
  |                                                          |
  |                    University of York                    |
  +---------------------------------------+

run name: run_name
NON: constructing arrays
NON: computing ground state density
NON: ground-state energy = 1.12463
NON: constructing arrays
NON: real time propagation
NON: N = 3, t = 1.00000, normalisation = 0.9999999999999843
NON: calculating current density
NON: t = 1.00000
LDA: E = 1.86638996 Ha, de = -4.450e-13, dn = 7.820e-13, iter = 19[K
LDA: ground-state energy: 1.8663899628889569
LDA: evolving through real time: t = 1.0990000000000144
LDA: calculating current density
LDA: t = 1.00000

EXT: constructing arrays
EXT: imaginary time propagation
EXT: t = 93.16000, convergence = 9.984857751219756e-121
EXT: ground-state converged
EXT: ground-state energy = 1.85031
EXT: constructing arrays
EXT: real time propagation
EXT: t = 0.45000
```

Non-interaction approximation

This is the simplest form of DFT, which acts as a baseline comparison to see whether an approximation, like the LDA, gives a good result compared to the non interacting result. The parameters are as follows:

- **non.rtol_solver** - Tolerance of linear solver in real time propagation (~e-13)
- **non.save_eig** - Save eigenfunctions and eigenvalues of Hamiltonian
- **non.RE** - Reverse-engineer non-interacting density
- **non.OPT** - Calculate the external potential for the non-interacting density

Local Density Approximation

This is the most common approximation used in DFT. The parameters are as follows:

- **lda.NE** - number of electrons used for LDA construction (1, 2, 3, 'heg')
- **lda.scf_type** - scf type (linear, pulay, cg)
- **lda.mix** - mixing parameter for linear and Pulay mixing (between 0 and 1)
- **lda.pulay_order** - history length for Pulay mixing (max: lda.max_iter)
- **lda.pulay_preconditioner** - preconditioner for Pulay mixing (None, kerker, rpa)
- **lda.kerker_length** - length over which density flunctuations are screened (Kerker)
- **lda.tol** - convergence tolerance in the density
- **lda.etol** - convergence tolerance in the energy
- **lda.max_iter** - maximum number of self-consistency iterations
- **lda.save_eig** - save eigenfunctions and eigenvalues of Hamiltonian
- **lda.OPT** - calculate the external potential for the LDA density

For the number of electrons, 'heg' is an acronym for 'homogeneous electron gas'. There are also the types of self-consistency available: 'linear', 'pulay' and 'cg'. 'Linear' is the least complicated and used in most situations. Density fluctuations can occur, which prevents LDA from reaching self-consistency. These different methods and mixing of methods will help reach self-consistency.

Exercise 2

Perform DFT calculations for 2 electrons with the LDA approximations using 'heg' and compare with the non-interaction approximation and exact calculation by plotting the electron densities.

[ ]:
```

```

Hartree-Fock calculation

The Hartree-Fock method is an alternative to DFT and is essentially a simplified version of many-body perturbation theory. The parameters are as follows:

- **hf.fock** - include Fock term (0 = Hartree approximation, 1 = Hartree-Fock approximation)
- **hf.con** - tolerance
- **hf.nu** - mixing term
- **hf.save_eig** - save eigenfunctions and eigenvalues of the Hamiltonian
- **hf.RE** - reverse engineering HF density
- **hf.OPT** - calculate the external potential from the HF density

Exercise 3

Compare the electron densities for a two electron system with the exact and non-interacting.

```
[ ]:
```

Reverse Engineering

The Reverse Engineering algorithm (for both time-independent and time-dependent systems) take the exact electron density and ‘reverse engineers’ the exact Kohn-Sham potential for the system. The parameters are as follows:

- **re.save_eig** - save Kohn-Sham eigenfunctions and eigen values of reverse-engineered potential
- **re.stencil** - discretisation of 1st derivative (5 or 7)
- **re.mu** - 1st convergence parameter in the ground-state reverse-engineering algorithm
- **re.p** - 2nd convergence parameter in the GS RE algorithm
- **re.nu** - convergence parameter in the time-dependent RE algorithm
- **re.rtol_solver** - tolerance of the linear solver in real-time propagation (~1e-12)
- **re.density_tolerance** - tolerance of the error in the time_dependent density
- **re.cdensity_tolerance** - tolerance of the error in the current density
- **re.max_iterations** - maximum number of iterations per time step to find the Kohn-Sham potential
- **re.damping** - damping factor used when filtering out noise in the Kohn-Sham vector potential (0: none)

Exercise

Reverse engineer the Kohn-Sham potential from the exact density from one of the previous exercises.

```
[ ]:
```

2.2 Getting Started - Further Details

This notebook will introduce all of the other parameters currently in the iDEA code. Note that the explanations are only very brief, so if a proper and full understanding is to be gained, study the relevent theory notebooks. There are elementary exercises at the end of each section to test understanding, with model answers at the end of the notebook. *They are considered elementary as they should only involve changing the values of relevent parameters and plotting the result*

```
[2]: # Setting up - same as previous notebook
from iDEA.input import Input
pm = Input.from_python_file('parameters.py')

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML
from iDEA.input import Input
from iDEA.results import Results as rs
```

2.2.1 Further Parameters

Exact Many-body Calculation

The exact many-body calculation is done by propagating the wavefunction in imaginary time, which eventually converges on the ground-state energy. The exact (ext) parameters are as follows:

- **ext.itol** - Tolerance of imaginary time propagation (~ 1e-12)
- **ext.itol_solver** - Tolerance of linear solver in imaginary time propagation (~ 1e-14)
- **ext.rtol_solver** - Tolerance of linear solver in real time propagation (~ 1e-12)
- **ext.itmax** - Total imaginary time
- **ext.iimax** - Imaginary time iterations
- **ext.ideltat** - Imaginary time step (DERIVED)
- **ext.RE** - Reverse engineer many-body density
- **ext.OPT** - Calculate the external potential for the exact density
- **ext.excited_states** - Number of excited states to calculate (0 for only ground-state)
- **ext.psi_gs** - Save the reduced ground-state wavefunction to file
- **ext.psi_es** - Save the reduced excited-state wavefunctions to file
- **ext.initial_psi** - initial wavefunction: qho, non, hf, lda1/2/3/heg, ext or wavefunction from previous run (e.g run_name)

Tolerance describes the difference between, say for example, densities between iterations before a successful convergence has occured.

Note that in the exercises, if a variable is not specified, such as electron number, take it to be the default value set in the parameters file. #### Exercise 1

Perfom exact many-body calculations for the ground state first excited state and calculate the relevent ELF. Display these results in a graph. *HINT: start by printing the parameters to find out what needs to be changed to get the desired result.*

[ ]:
```

```

Non-interaction approximation

This is the simplest form of DFT, which acts as a baseline comparison to see whether an approximation, like the LDA, gives a good result compared to the non interacting result. The parameters are as follows:

- **non.rtol_solver** - Tolerance of linear solver in real time propagation (~e-13)
- **non.save_eig** - Save eigenfunctions and eigenvalues of Hamiltonian
- **non.RE** - Reverse-engineer non-interacting density
- **non.OPT** - Calculate the external potential for the non-interacting density

Local Density Approximation

This is the most common approximation used in DFT. The parameters are as follows:

- **lda.NE** - number of electrons used for LDA construction (1, 2, 3, 'heg')

- **lda.scf_type** - scf type (linear, pulay, cg)
- **lda.mix** - mixing parameter for linear and Pulay mixing (between 0 and 1)
- **lda.pulay_order** - history length for Pulay mixing (max: lda.max_iter)
- **lda.pulay_preconditioner** - preconditioner for Pulay mixing (None, kerker, rpa)
- **lda.kerker_length** - length over which density flunctuations are screened (Kerker)
- **lda.tol** - convergence tolerance in the density
- **lda.etol** - convergence tolerance in the energy
- **lda.max_iter** - maximum number of self-consistency iterations
- **lda.save_eig** - save eigenfunctions and eigenvalues of Hamiltonian
- **lda.OPT** - calculate the external potential for the LDA density

For the number of electrons, 'heg' is an acronym for 'homogeneous electron gas'. There are also the types of self-consistency available: 'linear', 'pulay' and 'cg'. 'Linear' is the least complicated and used in most situations. Density fluctuations can occur, which prevents LDA from reaching self-consistency. These different methods and mixing of methods will help reach self-consistency.

Exercise 2

Perform DFT calculations for 2 electrons with the LDA approximations using 'heg' and compare with the non-interaction approximation and exact calculation by plotting the electron densities. *HINT: take a look at the parameters file directly if one cannot determine which parameter needs to be changed from the notebook's description of the parameters.*

```
[ ]:
```

Hartree-Fock calculation

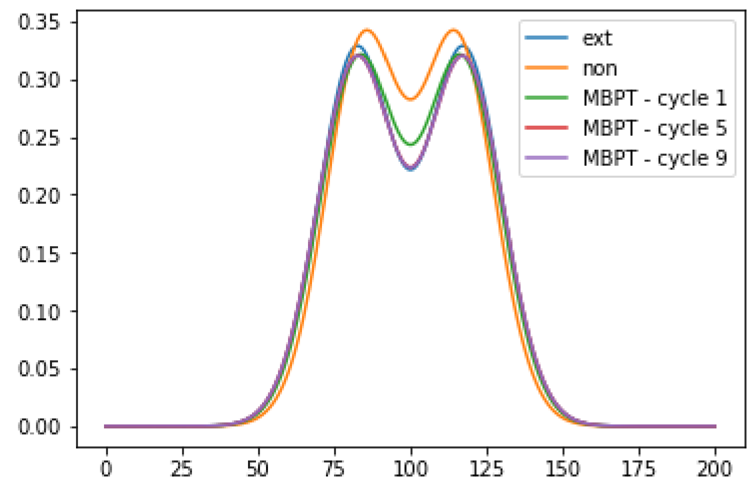
The Hartree-Fock method is an alternative to DFT and is essentially a simplified version of many-body perturbation theory. The parameters are as follows:

- **hf.fock** - include Fock term (0 = Hartree approximation, 1 = Hartree-Fock approximation)
- **hf.con** - tolerance
- **hf.nu** - mixing term
- **hf.save_eig** - save eigenfunctions and eigenvalues of the Hamiltonian
- **hf.RE** - reverse engineering HF density
- **hf.OPT** - calculate the external potential from the HF density

Exercise 3

Compare the electron densities for a two-electron system with the exact, non-interacting and HF. *HINT: each cycle will produce an output file. If ever in doubt of what your outputs are, take a direct look in **outputs/run_name/raw*.* (run_name being the default name)*

```
[4]:
```



Reverse Engineering

The Reverse Engineering algorithm (for both time-independent and time-dependent systems) take the exact electron density and 'reverse engineers' the exact Kohn-Sham potential for the system. The parameters are as follows:

- **re.save_eig** - save Kohn-Sham eigenfunctions and eigen values of reverse-engineered potential
- **re.stencil** - discretisation of 1st derivative (5 or 7)
- **re.mu** - 1st convergence parameter in the ground-state reverse-engineering algorithm
- **re.p** - 2nd convergence parameter in the GS RE algorithm
- **re.nu** - convergence parameter in the time-dependent RE algorithm
- **re.rtol_solver** - tolerance of the linear solver in real-time propagation (~1e-12)
- **re.density_tolerance** - tolerance of the error in the time_dependent density
- **re.cdensity_tolerance** - tolerance of the error in the current density
- **re.max_iterations** - maximum number of iterations per time step to find the Kohn-Sham potential
- **re.damping** - damping factor used when filtering out noise in the Kohn-Sham vector potential (0: none)

Exercise 4

Reverse engineer the Kohn-Sham potential from the exact density from one of the previous exercises. The exact densities are, in fact, the same for all the exercises so the result should be the same regardless. *HINT: make sure to set the reverse-engineering parameter to True.*

[ ]:

```

```

2.2.2 Model Answers

```
[ ]: #Exercise 1
     # As always set a run_name
     pm.run.name = 'exercise_1'

     # Turn on the correct parameters
     pm.run.NON = False
     pm.run.EXT = True
     pm.ext.elf_gs = True
     pm.ext.elf_es = True
     pm.ext.excited_states = 1

     # Start the simulation
     results = pm.execute()

     # Plotting the results
     ex1_results = rs.read("es_ext_elf1",pm)
     plt.plot(ex1_results, label="ELF - 1st excited state")
     plt.legend()
     plt.show()
```

```
[ ]: #Exercise 2
     # Setting up the parameters
     # run
     pm.run.name = 'exercise_2'
     pm.run.NON = True
     pm.run.LDA = True
     pm.run.EXT = True
     # sys
     pm.sys.NE = 2
     # lda
     pm.lda.NE = 'heg'

     # Starting the simulation
     results = pm.execute()

     # Plotting the results
     ex2_lda = rs.read("gs_ldaheg_den",pm)
     ex2_ext = rs.read("gs_ext_den",pm)
     ex2_non = rs.read("gs_non_den",pm)

     plt.plot(ex2_lda, label="lda - heg")
     plt.plot(ex2_ext, label="ext")
     plt.plot(ex2_non, label="non")
     plt.legend()
     plt.show()
```

```
[ ]: # Exercise 3
     # Setting up the parameters
     # run
     pm.run.name = 'exercise_3'
     pm.run.NON  = True
     pm.run.EXT  = True
     pm.run.HF   = True

     #results = pm.execute()

     # Plotting the results
```

(continues on next page)

(continued from previous page)

```
ex3_ext    = rs.read("gs_ext_den",pm)
ex3_non    = rs.read("gs_non_den",pm)
ex3_hf     = rs.read("gs_hf_den",pm)

plt.plot(ex3_ext, label="ext")
plt.plot(ex3_non, label="non")
plt.plot(ex3_hf,  label="hf")
plt.legend()
plt.show()
```

[ ]:
```
# Exercise 4
# Setting up the parameters

# Reverse-engineering the exact density from exercise 2, so need to be
# working out of the subdirectory containing that.
pm.run.name = 'exercise_2'
# turn off exact so that it won't rerun and overwite the previous data
pm.run.EXT = False
# turn off any unrelated parameters
pm.run.NON = False

# Turning on reverse-engineering
pm.ext.RE = True

#results = pm.execute()

# Plotting the results
ex4_RE    =  rs.read("gs_extre_vks",pm)
plt.plot(ex4_RE, label="reverse-engineered ext")

plt.legend()
plt.show()
```

2.3 Two electrons in a well

Now that's been quite a slog to get through all that theory, it'll be nice to bring the equations to life by actually running iDEA. We'll focus on the physics going on and won't worry too much about how iDEA is solving the equations under the hood.

2.3.1 Defining the external potential

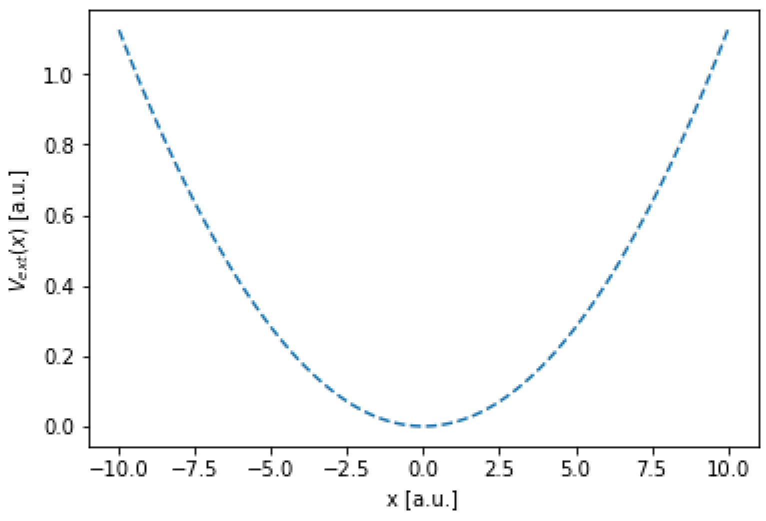
We start by investigating the electronic ground state of two electrons sitting in a harmonic well. Let's define our external potential $V_{ext}(r)$.

[1]:
```
# Let's define our external potential
omega = 0.15    # resonance frequency in atomic units
def harmonic_well(x):
    return 1/2 * omega**2 * x**2

# and plot it
import numpy as np
x = np.linspace(-10,10,100)
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
plt.plot(x, harmonic_well(x), '--')
plt.xlabel("x [a.u.]")
plt.ylabel(r"$V_{ext}(x)$ [a.u.]")
plt.show()
```



2.3.2 Computing the ground state

Now we'll fill $N = 2$ electrons into the system and use the iDEA code to find the exact solution to the time-independent Schrödinger equation

$$\left(-\frac{1}{2}\sum_i \frac{\partial^2}{\partial r_i^2} + V_{ext}(r_i) + \frac{1}{2}\sum_{i\neq j} v(r_i - r_j)\right)\Psi(r_1, r_2) = E\Psi(r_1, r_2)$$

```
[2]: from iDEA.input import Input
pm = Input()    # create input parameters with default settings
pm.sys.NE = 2  # we want 2 electrons
pm.sys.v_ext = harmonic_well  # pass on our external potential

print(pm.sys)  # show all system parameters

pm.run.name = "harmonic_well"
pm.run.EXT  = True      # run the exact system
pm.run.LDA  = True      # run DFT in the local density approximation
pm.run.NON  = True      # run the non-interacting approximation

print(pm.sys)  # show all run parameters
```

```
NE = 2
grid = 201
stencil = 3
xmax = 10.0
tmax = 1.0
```

(continues on next page)

(continued from previous page)

```
imax = 1001
acon = 1.0
interaction_strength = 1.0
im = 0
v_ext = <function harmonic_well at 0x10c942ae8>
v_pert = <function Input.__init__.<locals>.v_pert at 0x114d5dae8>
v_pert_im = <function Input.__init__.<locals>.v_pert_im at 0x114d321e0>

NE = 2
grid = 201
stencil = 3
xmax = 10.0
tmax = 1.0
imax = 1001
acon = 1.0
interaction_strength = 1.0
im = 0
v_ext = <function harmonic_well at 0x10c942ae8>
v_pert = <function Input.__init__.<locals>.v_pert at 0x114d5dae8>
v_pert_im = <function Input.__init__.<locals>.v_pert_im at 0x114d321e0>
```

We're going to compute the ground state both exactly and in the non-interacting and local-density approximations. Let's go!

[3]:
```
# run iDEA with specified parameters and save results
# this shouldn't take more than a minute
results = pm.execute()
```

```
                *    ****     *****       *
                     *   *    *          * *
                *    *    *   *         *   *
                *    *     *  *****    *     *
                *    *    *   *       *********
                *    *   *    *      *         *
                *    ****     ***** *           *

  +----------------------------------------+
  |          Interacting Dynamic Electrons Approach          |
  |              to Many-Body Quantum Mechanics              |
  |                                                          |
  |                      Release 2.2.0                       |
  |                                                          |
  |       Created by Piers Lillystone, James Ramsden,        |
  |       Matt Hodgson, Thomas Durrant, Jacob Chapman,       |
  |         Jack Wetherell, Mike Entwistle, Matthew          |
  |           Smith, Aaron Long and Leopold Talirz           |
  |                                                          |
  |                    University of York                    |
  +----------------------------------------+

run name: harmonic_well
NON: constructing arrays
NON: computing ground state density
NON: ground-state energy = 0.29996
LDA: E = 0.51502794 Ha, de = -9.249e-11, dn = 1.821e-12, iter = 23
```

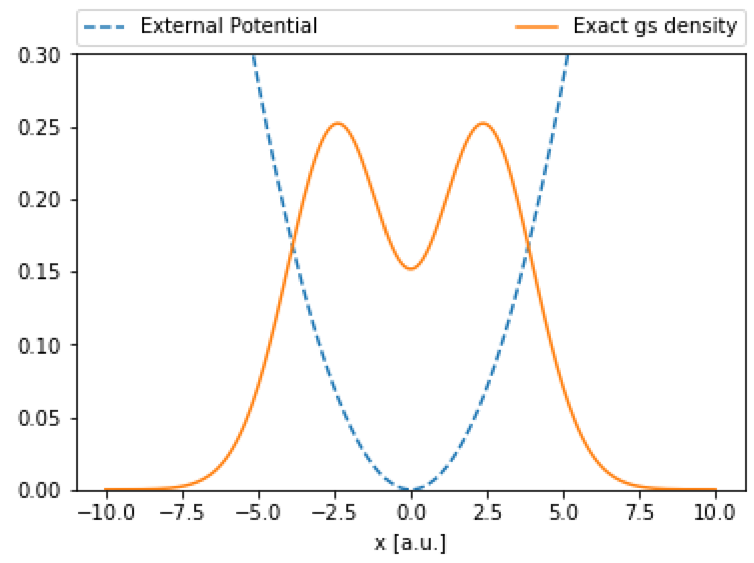
(continues on next page)

(continued from previous page)

```
LDA: ground-state energy: 0.5150279447967898
EXT: constructing arrays
EXT: imaginary time propagation
EXT: t = 141.88000, convergence = 1.0076922258026988e-11[K
EXT: ground-state converged
EXT: ground-state energy = 0.50441
all jobs done
```

Given $\Psi(r_1, r_2)$, we can compute the corresponding density $n(r)$ - in fact, iDEA has already done it for us.

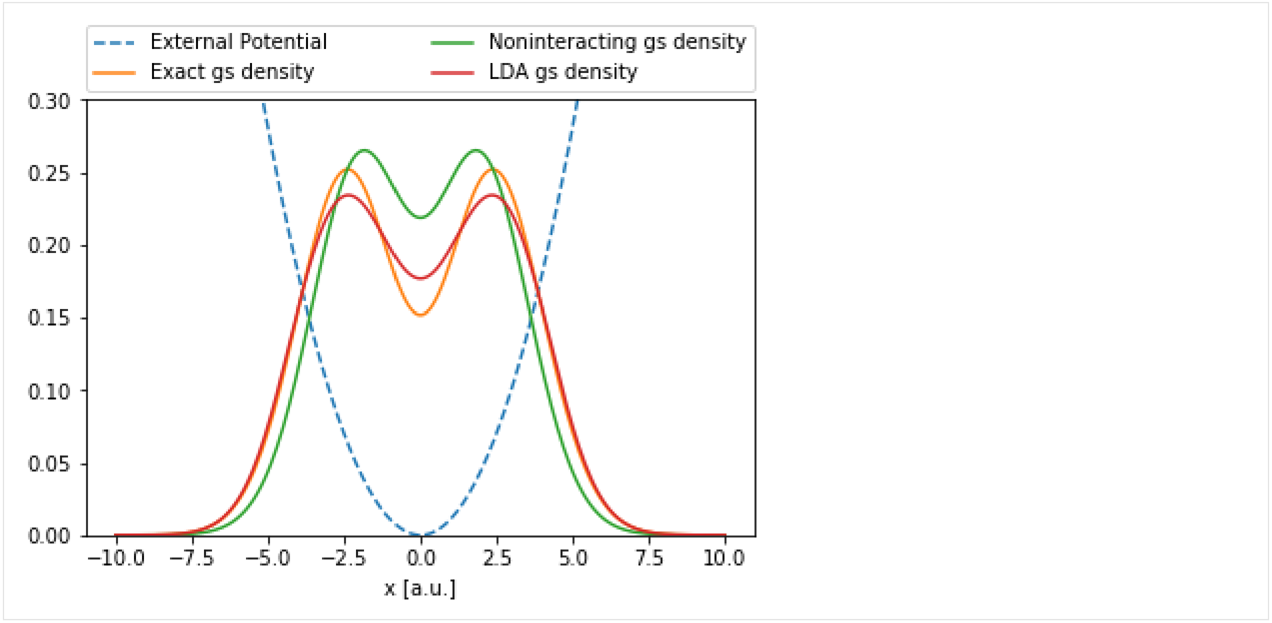
```
[4]: x = pm.space.grid
plt.plot(x,results.ext.gs_ext_vxt,'--', label = "External Potential")
plt.plot(x,results.ext.gs_ext_den, label = "Exact gs density")

plt.xlabel("x [a.u.]")
#plt.xlim([-5,5])
plt.ylim([0,0.3])
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
plt.show()
```



It turns out that for two (spinless) electrons, you get two bumps inside the well.

Let's see how the non-interacting and local-density approximations measure up against the exact solution.

```
[5]: plt.plot(x,results.ext.gs_ext_vxt,'--', label = "External Potential")
plt.plot(x,results.ext.gs_ext_den, label = "Exact gs density")
plt.plot(x,results.non.gs_non_den, label = "Noninteracting gs density")
plt.plot(x,results.lda.gs_lda2_den, label = "LDA gs density")

plt.xlabel("x [a.u.]")
#plt.xlim([-5,5])
plt.ylim([0,0.3])
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
plt.show()
```



This plot illustrates a few interesting things about the theory.

Firstly we can see that the non-interacting electron approximation gets the rough shape of the density right. However, the electron's are more likely to be found near the centre of the well since they prefer to be in regions of low potential, and don't have Coulomb interaction pushing them away from each other.

We also see that the LDA does a good job at predicting the *position* of density maxima, while still showing significant deviations in their *values*.

2.3.3 Reverse engineering

Now, in Kohn-Sham density functional theory, the task is to find the Kohn-Sham potential $V_{KS}(r)$ such that the solution of the non-interacting Kohn-Sham system

$$\left(-\frac{1}{2}\frac{\partial^2}{\partial r^2} + V_{KS}(r)\right)\psi_i(r) = i\frac{\partial}{\partial t}\psi_i(r)$$

yields the same density as the exact ground state. Since we are lucky enough to already *know* the exact $n(r)$ for our system, this task reduces to an optimisation problem.

Before we give this problem to iDEA, let's have a go at it ourselves. Given that the non-interacting electron density (green) is a little higher in the centre of the well than the exact density (red), the Kohn-Sham potential will probably need to add a bump in the centre of the well in order to push the Kohn-Sham electrons away.

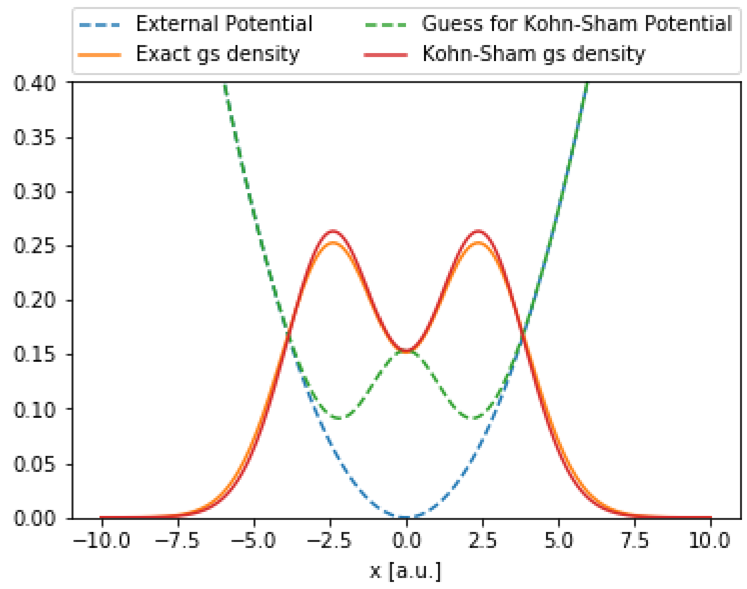
Try to optimize the Kohn-Sham potential to bring $n_{KS}(r)$ as close as possible to $n(r)$.

[32]:
```
alpha=0.2    # prefactor of Gaussian bump
sigma=1.3    # width of Gaussian bump
def kohn_sham_guess(x):
    return harmonic_well(x) + alpha / sigma * np.exp( -x**2 / (2*sigma**2))

# now we run the non-interacting system for this Kohn-Sham potential
pm.sys.v_ext = kohn_sham_guess
pm.run.EXT   = False
pm.run.LDA   = False
```

(continues on next page)

(continued from previous page)

```
pm.run.NON  = True
pm.run.verbosity = 'low' # suppress text output
results_non = pm.execute()  # this should be essentially instantaneous

plt.plot(x,results.ext.gs_ext_vxt,'--', label = "External Potential")
plt.plot(x,results.ext.gs_ext_den, label = "Exact gs density")

plt.plot(x,results_non.non.gs_non_vxt,'--', label = "Guess for Kohn-Sham Potential")
plt.plot(x,results_non.non.gs_non_den, label = "Kohn-Sham gs density")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)

plt.xlabel("x [a.u.]")
plt.ylim([0,0.4])
plt.show()

# computing the density-difference
den_diff = np.abs(results.ext.gs_ext_den - results_non.non.gs_non_den)
den_err = np.sum(den_diff) * pm.space.delta
print("Integrated absolute density difference: {:.3f}".format(den_err))
```



```
Integrated absolute density difference: 0.080
```

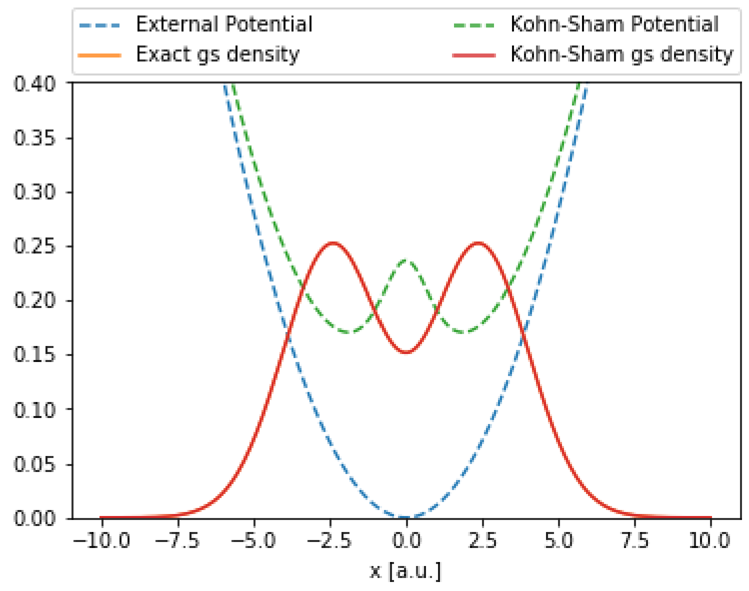
If you play around with the Kohn-Sham potential for a while, you will manage to get the solution pretty close to the exact one.

Now let iDEA have a go at finding the Kohn-Sham potential using its reverse engineering algorithm.

[7]:
```
# again, this should take less than a minute
pm.run.NON  = False
pm.ext.RE = True     # reverse engineer exact density
results_re = pm.execute()
print("Done")
```

```
Done
```

```
[26]: # re-plotting exact results from before
plt.plot(x,results.ext.gs_ext_vxt,'--', label = "External Potential")
plt.plot(x,results.ext.gs_ext_den, label = "Exact gs density")

# align external and Kohn-Sham potential a the box edge
vks = results_re.extre.gs_extre_vks
vks += results.ext.gs_ext_vxt[0] - vks[0]

# adding results from reverse-engineering
plt.plot(x,vks,'--', label = "Kohn-Sham Potential")
plt.plot(x,results_re.extre.gs_extre_den, label = "Kohn-Sham gs density")

plt.xlabel("x [a.u.]")
plt.ylim([0,0.4])
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
plt.show()

# computing the density-difference
den_diff = np.abs(results.ext.gs_ext_den - results_re.extre.gs_extre_den)
den_err = np.sum(den_diff) * pm.space.delta
print("Integrated absolute density difference: {:.10f}".format(den_err))
```



```
Integrated absolute density difference: 0.0000000001
```

If everything has gone right, the exact ground-state and Kohn-Sham density should coincide exactly.

You may notice that even though we have aligned V_{KS} and V_{ext} at the edge of the box, **continue from here**

2.4 Two electrons in a double well

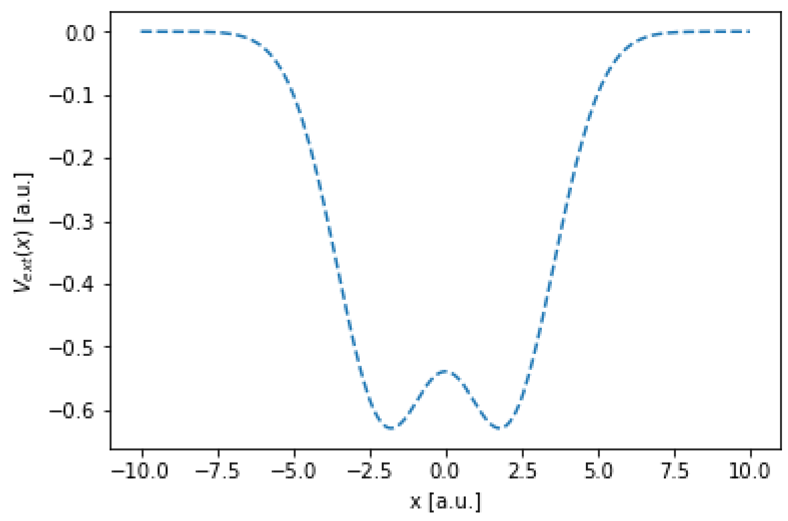
We now encourage you to have a go yourself. A nice system to look at to explore these ideas is a symmetric double well of varying separations. We'll get you started with your first potential, but then it's up to you to run the code yourself.

Your goal is to

- plot the exact, non-interacting and LDA electron densities for two (or more if you'd like!) systems of symmetric double wells of varying separations
- see how the densities compare, and
- explain why does the quality of the approximations depends on the separation

Note: You may want to increase your grid size, xmax.

[1]:
```
####From below here, this wouldn't be entered in the notebook you'd first open the␣
↪notebook##
import numpy as np

def symmetric_double_well_separation1(x):
    return -0.6 * np.exp(-0.2 * ((x-2)**2)) -0.6 * np.exp(-0.2 * (x+2)**2)

import matplotlib.pyplot as plt
x = np.linspace(-10,10,201)
plt.plot(x,symmetric_double_well_separation1(x),'--',label="External Potential")
plt.xlabel("x [a.u.]")
plt.ylabel(r"$V_{ext}(x)$ [a.u.]")
plt.show()
```



[2]:
```
from iDEA.input import Input
pm = Input()
pm.run.name = "symmetric_well_run1"
pm.run.LDA = True
pm.run.verbosity = "low"
pm.sys.grid = 201
pm.sys.xmax = 20.0
pm.sys.v_ext = symmetric_double_well_separation1

#Checking that we've got everything turned on that we want
print(pm.run)
print(pm.sys)
```

```
name = 'symmetric_well_run1'
time_dependence = False
verbosity = 'low'
```

(continues on next page)

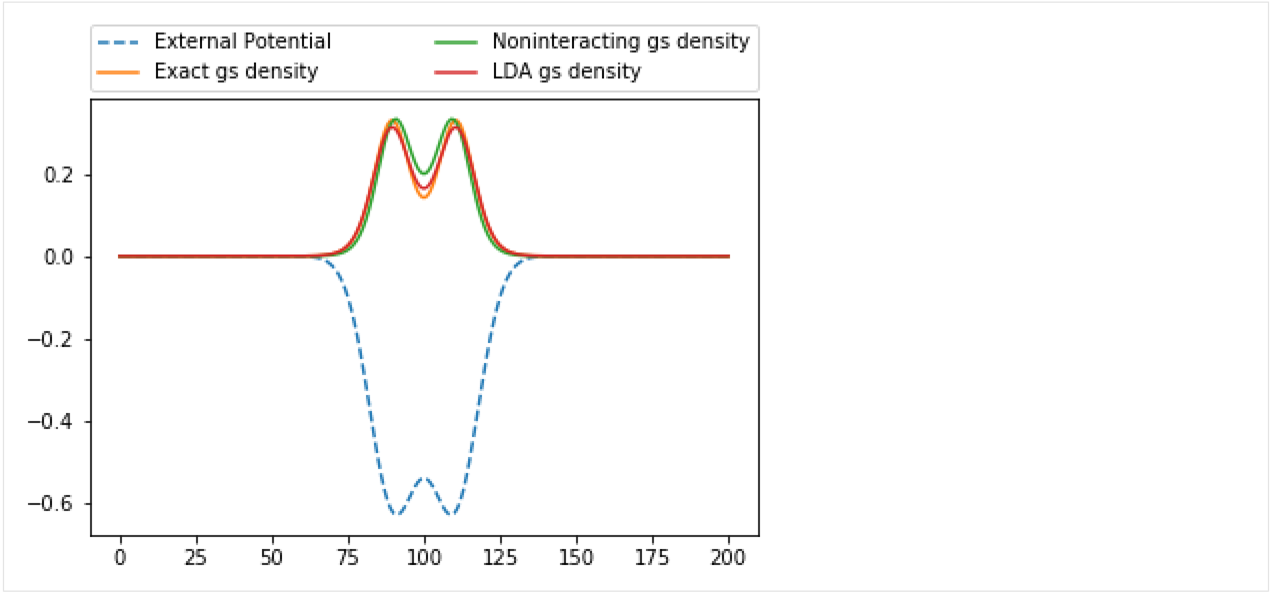
(continued from previous page)

```
save = True
module = 'iDEA'
NON = True
LDA = True
MLP = False
HF = False
EXT = True
MBPT = False
HYB = False
LAN = False

NE = 2
grid = 201
stencil = 3
xmax = 20.0
tmax = 1.0
imax = 1001
acon = 1.0
interaction_strength = 1.0
im = 0
v_ext = <function symmetric_double_well_separation1 at 0x109bcebf8>
v_pert = <function Input.__init__.<locals>.v_pert at 0x115133400>
v_pert_im = <function Input.__init__.<locals>.v_pert_im at 0x115146620>
```

[3]:
```
results = pm.execute()
print("Done")
```

```
Done
```

[4]:
```
import matplotlib.pyplot as plt

plt.plot(results.ext.gs_ext_vxt, '--', label = "External Potential")
plt.plot(results.ext.gs_ext_den, label = "Exact gs density")
plt.plot(results.non.gs_non_den, label = "Noninteracting gs density")
plt.plot(results.lda.gs_lda2_den, label = "LDA gs density")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
plt.show()
```



```
[ ]:
```

2.5 Exercise: Many electron tunnelling systems

Having read notebooks 1-4, you should be in a position to tackle this exercise. It is designed to test aspects from each of the topics you have learned so far. A worked solution to this exercise is provided, with a small amount of explanation of the underlying physics, but we highly encourage you to fully attempt this exercise first before consulting the solutions. There truly is no better way to check you're up to speed with everything you ought to be than by having a go yourself!

The system of interest is going to be a double antisymmetric well, with the exact form of the potential being

$$-\frac{6}{5}exp\left(\frac{-1}{125}(x-10)^4\right) - \frac{9}{10}exp\left(\frac{-1}{10}(x+10)^2\right),$$

or in python format

```
[2]: # -(6.0/5.0)*math.e**(-(1.0/125.0)*(x-10)**4)-(9.0/10.0)*math.e**(-(1.0/10.
     ↪0)*(x+10)**2)
```

Now we'd like you to ** run iDEA, with time dependence and reverse engineering switched on for the non-interacting, exact and LDA cases.** Be sure to adjust your system parameters to suitable values to ensure convergence and to include the whole of the potential. We also recommend running the time dependence for at least $t_{max} = 10 $ a.u. to ensure you are able to see the features we're interested in. The perturbation applied at $t = 0 $ is just the default linear electric field so there is no need to change this for the time being (but you're more than welcome to adjust this and see what its effect is).

After iDEA has finished running, plot your ground state densities for the each of the above cases and the external potential on the same graph commenting on any differences.

```
[4]: # Import iDEA and parameters
```

[5]:
```
# Adjust the parameters to answer the exercise
# NB - if your system has trouble running iDEA from the notebook, try setting pm.run.
↪verbosity to 'low'
```

[6]:
```
# Change the external potential to that given above
```

[7]:
```
# Optional (but recommended) - check that parameters file has updated as you expected
```

[8]:
```
# Hopefully everything is good to go now, so go ahead and run iDEA
```

Hopefully everything has worked as expected so far. Getting to this point means you're comfortable with the basics of running the iDEA code. If you're having problems with this, go back and read the "Getting Started with iDEA" notebook.

[9]:
```
# Import matplotlib and plot each of the densities you've calculated thus far
```

Use this cell to comment on the differences you see in the graphs:

That's the first part done! Now let's have a look at why DFT has done so well. **Plot the reverse engineered exact KS potential and the external potential on the same graph and comment on the major differences**

[10]:
```
# Plot the KS potential and the external potential on the same graph
```

Use this cell to comment on the major differences between between the two potentials:

Now we're ready to move onto looking at the time dependence of the system. **Animate how the exact density changes over time.**

[11]:
```
# Import the relevant packages for animations
```

[1]:
```
# Store your time dependent solution as an array
```

[2]:
```
# Set the background for the animation
```

[3]:
```
# Define the init function
```

[4]:
```
# Define the animation function
```

[5]:
```
# Display the animation
```

Use this cell to comment of the major features of the time-evolution. How can we tell it is a tunnelling system?

[ ]:

2.6 Convergence with iDEA

2.6.1 Introduction

In the many-body interacting quantum particle system, there are what is known as convergence parameters. As the wavefunction is propagated, these values need to 'settle' on a result before they can be considered useful; they need to be slightly adjusted either way to assess their stability.

2.6.2 How to converge

To test whether the parameter in question has been conserved, it needs to be reasonably concluded that after adjusting the initial parameters ever so slightly that the final result does not change a significant amount.

In this notebook, we will use the example of the total energy to test convergence. Firstly, we need to set up the environment, as normal:

[2]:
```
from iDEA.input import Input
# import plotting software
import matplotlib.pyplot as plt
# import parameters file into an object
pm = Input()
pm.run.NON = True
pm.run.name = "convNotebook"
pm.check()
```

Create a loop which runs the code for different values of xmax.

[3]:
```
pm.run.verbosity = 'low'
# Converging xmax parameter
for xmax in [4,6,8,10]:
    # Note: the dependent sys.deltax is automatically updated
    pm.sys.xmax = xmax
    pm.sys.grid = 401

    # perform checks on input parameters
    pm.check()
    results = pm.execute()
    E = results.non.gs_non_E
    print(" xmax = {:4.1f}, E = {:6.4f} Ha".format(xmax,E))
```

```
 xmax =  4.0, E = 0.5733 Ha
 xmax =  6.0, E = 0.5015 Ha
 xmax =  8.0, E = 0.5000 Ha
 xmax = 10.0, E = 0.5000 Ha
```

The Energy has remained the same for the last two outputs and can be said to have converged.

[ ]:

CHAPTER 3

Theory

3.1 Many-electron Quantum Mechanics

3.1.1 Motivation

The theory of interacting electrons is extremely rich and complex. This, along with recent advances in computing capabilities, means that more than 10,000 papers are published on electronic structure theory each year. This research is yielding novel understanding in a vast range of fields including physics, chemistry and materials science.

This is because, on the atomic level, electrons are the *glue* of matter and if we understand how electrons move in their environment *and* how they interact with each other, we can predict the electronic, optical and mechanical properties of materials.

3.1.2 Notation

We are going to use Hartree atomic units where $e = \hbar = m_e = 4\pi\varepsilon_0 = 1$ which saves a lot of clutter! This means the standard unit of length is the Bohr radius $a_0 = 5.29 \times 10^{-11}\mathrm{m}$, and the unit of energy is the Hartree $E_H = 2\mathrm{Ry} = 27.2\mathrm{eV}$. Also be aware that capital Ψ refers to a many-body wave function whereas lower case ψ refers to single particle wave functions.

Below, we keep everything in 3 dimensions to keep things general, but be aware that iDEA works in 1D only.

3.1.3 Schrödinger equation

The name of the game is to solve the Schrödinger equation for both the ground state system, $\hat{H}\Psi = E\Psi$ and for the time dependent one, $\hat{H}\Psi = i\frac{\partial\Psi}{\partial t}$ (we'll consider the time dependent case later).

In an ideal world, we would solve the many-body Schrödinger equation exactly and then armed with these wavefunctions, we'd have complete knowledge of the system and be able to make precise predictions. Unfortunately, however, the problem is much too hard to solve in the way. Let's have a look at the Hamiltonian to see why.

$$\hat{H} = -\frac{1}{2}\sum_i \nabla_i^2 + V_{ext}(\mathbf{r}_i) + V_{ee}(\mid \mathbf{r}_i - \mathbf{r}_j \mid),$$

where we have used the Born-Oppenheimer approximation (treating the nuclei as so massive that they do not move on the timescale of electron motion), and the fact that the nuclei-nuclei interacting is constant and so we simply shift our zero of energy to absorb that term. V_{ext} is the external potential in which the electrons move, in the case of electrons moving in a potential set up by the nuclei, $V_{ext} = -\sum_{i,I}\frac{Z_I}{|\mathbf{r}_i - \mathbf{R}_I|}$. V_{ee} is the electron-electron interaction which in 3 dimensions takes the form $V_{ee} = \frac{1}{2}\sum_{i\neq j}\frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}$, but of course has a different 1D form which is implemented in the iDEA code.

The difficulty arises with the third term, the electron-electron interaction. This term means that the Schrödinger equation isn't seperable so the wavefunction isn't simply a suitable anti-symmetric product of single particle wavefunctions. We now explore several approaches to solve this problem that are used by the iDEA code.

3.1.4 Exact solution

We are going to pretend from here that our electrons are spinless, so we can ignore any spin-orbit effects. This also has more profound impacts on the universe we're considering. In a spinful universe, only electrons of equal spins would feel the exchange interaction, but in our spinless universe all electrons feel it. This means we get to see the effects of V_{xc} for systems with a smaller number of electrons so we get to maximise the amount of physics we can highlight for a given effort (on both your part and mine!).

As we mentioned earlier, the exact problem is, normally, much too hard to solve; the difficulty growing exponentially with the number of electrons in the system. However, for systems with 2 or 3 electrons, we *are* able to solve the problem exactly. In fact, this is one of the key concepts in iDEA - solving simple systems exactly to allow us to compare, and possibly improve, the approximate solutions. Armed with these improvements, we are in a better position to tackle the larger systems with many electrons.

Given we can't solve the many-electron system exactly in general, we need to have a look at the various different approaches of tackling the problem in more complicated systems.

3.1.5 Complete neglect of interaction

This is by far the simplest approach - you just completely ignore the Coulomb interaction between the electrons. Clearly this massively simplifies the problem, giving a separable Hamiltonian for a start! This means that we are able to solve the Schrödinger equation by the method of separation of variables, solving it for each electron individually. Then the total wavefunction is just a product of these single particle wave functions, in a suitable anti symmetric arrangement. One often constructs this wavefunction with something known as a "Slater determinant":

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, \ ..., \mathbf{r}_N) = \frac{1}{\sqrt{N!}}\begin{vmatrix} \psi_1(\mathbf{r}_1) & \psi_1(\mathbf{r}_2) & \dots & \psi_1(\mathbf{r}_N) \\ \psi_2(\mathbf{r}_1) & \psi_2(\mathbf{r}_2) & \dots & \psi_2(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_N(\mathbf{r}_1) & \psi_N(\mathbf{r}_2) & \dots & \psi_N(\mathbf{r}_N) \end{vmatrix}.$$

Unfortunately, but unsurpringsingly, making this approximation means that you lose a lot of the physics in the problem (look at the double antisymmetric well notebook for a prime example). Generally, in this system electrons spend more time closer to each other than they would if the repulsion between them had been taken into account. This is often a useful thing to calculate, however, since the code runs quickly and can then provide a reasonable first guess for the density for a self-consistant field approach - more on this later.

3.2 Density Functional Theory (DFT)

A functional is a function of a function. This may sound like a scary word, but you've definitely worked with them before! (e.g. an integral is a functional $\int f(x)dx$). The notation we're going to use for functionals is square brackets, e.g. F[x] means F is a function of x which is in turn a function of another variable, so F is a functional.

DFT is used to calculate the ground state of a system, and so is independant of time. The most important function in DFT is the electron density, $n(\mathbf{x})$. Once you've got the density, you are able to calculate anything you want about the system (more on this later). This vastly simplifies the problem, since the density is only a function of 3 varibles, opposed to the 3N varibles of the full many body wavefunction. The simplest formulations of DFT treat the electron-electron interaction through a mean-field only approach, that is reducing the many-body, interacting electron problem into many non-interacting electrons moving in an effective potential.

3.2.1 Aside - Mean Field Theories

If you are familiar with the concept of a mean field theories, you might like to think of DFT in this way but it isn't necessary to understand the following. We'll include a quick reminder here of mean field theory here in case you'd like to think of DFT in this way. One can think of this as replacing operators by their expectation values, (or more rigorously, writing an operator as its mean plus fluctuations about the mean and then taking the 0th order term, $\hat{O} = \langle\hat{O}\rangle + \delta\hat{O} \approx \langle\hat{O}\rangle$.) Clearly this means that the mean field has no fluctuations. This greatly simplifies the problem since you don't need to keep track of each of the individual particles and instead just one particle moving in the mean field. (Add things about SCF)

3.2.2 Hohenberg-Kohn Theorems

DFT is built on two main theorems - the Hohenberg-Kohn theorems *[Hohenberg1964]* - which we will state without proof here in the interests of brevity, but we very much encourage you to look into their proofs - they're really not so bad.

From this point on, we are going to specialise into 1 dimension to be faithful to the iDEA code, but these ideas are easily generalised to 3D.

Theorem 1

The external potential is a unique functional of the electron density only (up to an additive constant). So the Hamiltonian, and therefore all ground state properties, are determined solely by the electron density.

This is a very far reaching statement! Once we've got the electron density, we have got everything we could want to know about the system. It is important to recall how the density is related to the many-body wave function:

$$n(\mathbf{x}) = N\int dx_2 \int dx_3 \,...\int dx_N \mid \Psi(x, x_2,\ ..., x_N) \mid^2$$

where the prefactor of N is included to account for the arbitrary assignment of which of the electrons hasn't had its coordinate integrated over.

Theorem 2

The ground state energy may be obtained variationally, and the density which minimises this energy is the exact ground state energy.

NB There is a nuance here, but a very important one! During the proofs of these theorems, one assumes that the electrons are in their ground states and so DFT is only valid for ground state systems.

Now, taking these two theorems together prove that a universal functional must exist, but sadly don't even give us a hint as to what it should look like, (or even how to calculate the ground state energy). Indeed, there are no known exact functionals for systems of more than one electron! Also, you shouldn't get too excited by the electron density being the central parameter, as ever things are more complicated than they seem. Although *in principle* it is possible to determine all properties of the the system from $n(\mathbf{x})$, in practice it isn't that easy. The is reason is we often don't know *how* to go from $n(\mathbf{x})$ to the quantity we're interested in finding and so have to revert back to the set of N wavefunctions.

At this point we pick up the *Kohn-Sham* formulation of DFT *[Kohn1965]*, which is what opened the door for so much progress in this field.

3.2.3 Kohn-Sham DFT

In the Kohn-Sham (KS) formulation of DFT, instead of considering the full system of N interacting electrons, we instead look at a fictitious system of N non-interacting electrons moving in an effective KS potential, V_{KS}. The single-particle KS orbitals are constrained to give the same electron density as that of the real system, so we can then, in theory at least, use theorem 1 to find out anything we want about the system. This (KS density yielding the exact density) is actually an assumption of the KS construction of the fictitious system as no rigorous proofs for realistic systems. other properties of the KS system are *not* the same as the real system (e.g. the kinetic energy of the auxiliary system won't, in general, be the same as that of the real system).

Recall that we are treating the electrons as spinless, so we constrain the system such that there is only one electron per KS orbital which gets around any possible problems arising from the Pauli exclusion principal.

We're going to brush over a few of the formalities here, since, for example, knowledge of how to use Lagrange multipliers to ensure particle conservation doesn't give greater insight into the physics.

The goal from here is to solve the Schrödinger-like equation

$$\left(-\frac{1}{2}\frac{d^2}{dx^2} + V_{KS}(x)\right)\psi_i(x) = \varepsilon_i\psi_i(x),$$

where V_{KS} is the Kohn-Sham potential that we'll discuss shortly, and ε_i are the single-partle eigenvalues. It is worth emphasising that this equation is, in principal, exact.

The KS potential is given by

$$V_{KS}(x) = V_{ext}(x) + V_H(x) + V_x(x) + V_c(x),$$

where V_{ext} is the external potential arising from the electron's interaction with the nuclei, V_H is the Hartree potenial, V_x is the exchange potential and V_c the correlation potential. The last two terms are often lumped together into one exchange-correlation potential, V_{xc}, which we shall use fron now on.

3.2.4 Hartree potential

The easiest way to understand the origin of the Hartree potential is to imagine freezing all the electrons in space, and then seeing what the electrostatic potential is due to these electrons.

$$V_H(x) = \int n(x')u(\mid x - x' \mid)dx',$$

where u is the softened coulumb interaction implemented in iDEA. If you examine definition of the Hartree potential you'll notice that it includes self-intreaction, that is electron's interacting with other parts of their *own* charge densities - don't worry, this gets accounted for later. It's worth taking a moment here to reflect on where we are so far. On the face of things, it might seem like we have everything we need to solve this problem exactly. We've entirely accounted for the Coulomb interaction between the electrons and the nuclei and between the electrons themselves. So why do we need to bother including V_{xc}? The reason is that in defining V_H, we froze the electrons in place and got an *electrostatic* potential, but of course the electrons in a real system will be moving, and it is this movement that gives rise to the exchange-correlation potential.

3.2.5 Exchange-Correlation potential

The origin of the exchange part of the potential is due to the exchange symmetry of the wavefunction of the system of identical particles (we'll restrict our treatment to fermions here). When fermions get close to each other they experience "Pauli repulsion", which causes the expectation values between them to be larger. So when the electrons are moving in the sample, they stay further away from each other than one would naively expect. The correlation of the system is a bit harder to put on explicit physical basis but it is a measure of how much the motion of one electron affects that of another. V_{xc} also corrects for the self interaction in the Hartree potential.

The problem is that we don't know what the exchange-correlation functional looks like for any system more complicated than the homogenous electron gas (HEG), which is where KS DFT goes from being an exact theory to an approximate one. We'll discuss one of these approximations later.

DFT's strength lies in the fact that V_{xc} is a relatively small contribution to V_{KS} so this term only being approximately correct doesn't change the form of the KS potential too drastically, which gives accruate KS oribitals and hence the electron density given by

$$n(x) = \sum_i \mid \psi_i(x) \mid^2 .$$

The alert reader may notice a problem here. We need the KS oribitals to get the density by the above equation. To get the orbitals we need to solve the Schrödinger-like equation, however, that requires knowledge of the KS potential, which in turn depends on the electron density of the system. So to solve this we put in a guess of the electron density (often the density obtained from the non-interacting electron approximation), then plug this into the Schrödinger-like equation for the orbitals and then get the density from those. You then compare this new density with the old one. If there has been a change, we plug this new density in and try again. We keep this iteration going until we reach a *self consistent solution*, or in practice that the change from the old density to the new one is very small.

Of course this all assumes we know the form of the KS potential, but as we mentioned earlier, no one knows the form of the exchange-correlation functional which stops us doing this calculations exactly. One of the most common approximations is to use the *local density approximation* (LDA).

3.2.6 Local Density Approximation (LDA)

In the LDA, the functional only depends on the place where we are evaluating the density (hense the 'local' part of its name). The energy functional is given by

$$E_{xc}^{LDA}[n(x)] = \int \varepsilon_{xc}^{HEG}(n)\, n(x)\, dx,$$

where $\varepsilon_{xc}^{HEG}(n)$ is the exchange-correlation energy per particle for the homogenous electron gas. Armed with this functional, we can get V_{xc}^{LDA} by using a functional derivative, which is written as

$$V_{xc}^{LDA} = \frac{\delta E_{xc}^{LDA}}{\delta n}.$$

Once, we have V_{xc}^{LDA}, we can get the KS potential and go through the process of finding a self consistent solution.

References

3.3 Reverse engineering

One of the great abilities of iDEA is being able to obtain the *exact* KS potential. This is done by solving the Schrodinger equation exactly, obtaining the exact electron density. From here, the iDEA code then works backwards to see what KS potential would have given this electron density (more details are given here). The reason for doing this is that it allows us to compare the exact reverse-engineered KS potential with that given by approximations, such as the LDA - using this comparison to improve the approximations we're using.

On these pages we are going to briefly introduce the theory behind the physics that the iDEA code is trying to unravel. We aim to cover all the essentials, but if you're looking for more complete treatments, we strongly recommend the following two books:

1. Interacting Electrons: Theory and Computational Approaches by Martin, Reining and Ceperley
2. Electronic Structure: Basic Theory and Practical Methods again by Martin

The second book goes into a little more detail on the basics than the first, so might be a better starting point for beginners.

CHAPTER 4

Implementation

Here we are going to cover the basics of how the different modules of iDEA are implemented.

4.1 What's the big iDEA?

The iDEA code has been developed to explore small model systems of interacting electrons, in order to gain insight into the crucial features present in time-dependent many-electron systems. This is allowing us to tackle important challenges in this area, e.g. developing an accurate description of electronic quantum transport - time-dependent electrical currents carried by electrons in response to an applied voltage - through nanostructures and nanodevices.

We use exact solutions of the many-electron time-dependent Schrödinger equation to study the performance of non-equilibrium Green's-function theory and time-dependent density-functional theory (TDDFT), along with a powerful reverse-engineering technique which allows us to calculate the exact exchange-correlation functionals in TDDFT.

4.2 EXT (Exact)

The EXT code solves the many-electron time-independent Schrödinger equation to calculate the fully correlated, ground-state wavefunction for a one-dimensional finite system of 2 or 3 spinless electrons interacting via the softened Coulomb repulsion $(|x-x'|+1)^{-1}$. A perturbing potential is then applied to the ground-state system and its evolution is calculated exactly through solving the many-electron time-dependent Schrödinger equation.

4.2.1 Calculating the ground-state

An arbitrary wavefunction Ψ is constructed (preferably close to the ground-state of the system) and then propagated through imaginary time using the Crank-Nicolson method. Using the eigenstates of the system $\{\psi_m\}$ as a basis:

$$\Psi(x_1, x_2, \ldots, x_N, \tau) = \sum_m c_m e^{-E_m \tau} \psi_m.$$

Providing the wavefunction remains normalised, the limiting value is the ground-state of the system:

$$E_{m+1} > E_m \;\; \forall\, m \in \mathbb{N}^0 \implies \lim_{\tau\to\infty} \Psi(x_1, x_2, \ldots, x_N, \tau) = \psi_0.$$

4.2.2 Time-dependence

A perturbing potential is applied to the Hamiltonian, $\hat{H} = \hat{H}_0 + \delta\hat{V}_{\text{ext}}$. The system is initially in its ground-state and its evolution is calculated by propagating the ground-state wavefunction through real time using the Crank-Nicolson method.

4.2.3 One-electron systems

EXT also works for systems of 1 electron. Unlike for 2 or 3 electron systems, the ground-state is calculated using an eigensolver. When a perturbation is applied to the system, its evolution is calculated by propagating the ground-state wavefunction through real time using the Crank-Nicolson method (like in the 2 or 3 electron systems).

4.3 RE (Reverse-Engineering)

The RE code calculates the exact Kohn-Sham (KS) potential [and hence exact exchange-correlation (xc) potential] for a given electron density $n(x, t)$.

4.3.1 Ground-state Kohn-Sham potential

The ground-state KS potential $V_{\text{KS}}(x, 0)$ is calculated by starting from the unperturbed external potential and iteratively correcting using the algorithm:

$$V_{\text{KS}}(x, 0) \to V_{\text{KS}}(x, 0) + \mu[n_{\text{KS}}(x, 0)^p - n(x, 0)^p],$$

where $n_{\text{KS}}(x, 0)$ is the ground-state KS electron density, and μ and p are convergence parameters. The correct $V_{\text{KS}}(x, 0)$ is found when $n_{\text{KS}}(x, 0) = n(x, 0)$.

4.3.2 Time-dependent Kohn-Sham potential

The time-dependent KS potential $V_{\text{KS}}(x, t)$ is calculated by applying a temporary gauge transformation and iteratively correcting a time-dependent KS vector potential $A_{\text{KS}}(x, t)$ using the algorithm:

$$A_{\text{KS}}(x, t) \to A_{\text{KS}}(x, t) + \nu \left[\frac{j_{\text{KS}}(x, t) - j(x, t)}{n(x, t) + a}\right],$$

where $j(x, t)$ is the current density of the interacting system and $j_{\text{KS}}(x, t)$ is the current density of the KS system. Once the correct $A_{\text{KS}}(x, t)$ is found, the gauge transformation is removed to calculate the full time-dependent KS potential as a scalar quantity.

4.4 LDA (Local Density Approximation)

The LDA is the most common approximation to the exchange-correlation (xc) functional in density functional theory (DFT). The LDA code implements 4 different functionals that we have developed *[Entwistle2018]*- 3 were constructed from slab-like systems of 1, 2 and 3 electrons, and the other was constructed from the homogeneous electron gas (HEG), with our softened Coulomb interaction. These approximate xc potentials allow us to approximate the electron density of systems, for comparison with exact solutions.

4.4.1 Calculating the ground-state

As an initial guess, the Kohn-Sham (KS) potential V_{KS} is approximated as the external potential V_{ext}. From this a set of non-interacting orbitals are computed through solving the KS (single-particle Schrödinger) equations:

$$\{\phi_j, \varepsilon_j\},$$

and from these the density $n(x)$ is calculated.

Using this density, an approximate xc potential V_{xc} is constructed using the chosen LDA functional. From this V_{KS} is refined, through a conjugate gradient method, Pulay mixing or linear mixing. For example, in linear mixing:

$$V_{\mathrm{KS}}^{i+1}(x) = (1-\alpha)V_{\mathrm{KS}}^{i}(x) + \alpha V_{\mathrm{KS}}^{\mathrm{LDA}}(x),$$

where the Kohn-Sham potential at the current iteration V_{KS}^{i} and the Kohn-Sham potential constructed using the LDA $V_{\mathrm{KS}}^{\mathrm{LDA}}$ are mixed to generate the Kohn-Sham potential at the next iteration V_{KS}^{i+1}.

These steps are repeated until we reach self-consistency, i.e. $V_{\mathrm{KS}}(x)$ and $n(x)$ are unchanging.

4.4.2 Time-dependence

After an approximate ground-state V_{KS} is found, the perturbing potential is applied to the ground-state Hamiltonian, $\hat{H} = \hat{H}_0 + \delta\hat{V}_{\mathrm{ext}}$. The system's evolution is calculated by propagating the ground-state KS orbitals through real time using the Crank-Nicolson method, and applying the LDA adiabatically (ALDA).

4.4.3 References

4.5 HF (Hartree–Fock)

The HF code solves the Hartree–Fock equation to approximately calculate the ground state density of a one-dimensional finite system of N like-spin electrons, with Hamiltonian $\hat{T} + \hat{U} + \hat{V}_{\text{ext}}$, where $\hat{U}$ is the operator for the softened Coulomb interaction potential given by

$$u(x,y) = (1 + |x-y|)^{-1}\,.$$

A perturbing potential may be applied to the ground state system, and the time-dependent Hartree–Fock equation solved approximately to calculate the system's time evolution.

4.5.1 Calculating the ground state

Orbitals and corresponding non-degenerate energy eigenvalues $\{\varphi_j, \varepsilon_j\}_{j=1}^{N}$ for a non-interacting system of N electrons are first calculated from the single-particle Schrödinger equation with Hamiltonian $\hat{T} + \hat{V}_{\text{ext}}$.

We then proceed to calculate the ground state electron density of this system:

$$n(x) = \sum_{j=1}^{N} |\varphi_j(x)|^2\,.$$

From this we find the Hartree potential

$$v_{\mathrm{H}}(x) = \int n(y)u(x,y)\,\mathrm{d}y\,,$$

and the Fock matrix (the self-energy in the exchange-only approximation)

$$\Sigma_{\text{x}}(x,y) = -\sum_{j=1}^{N} \varphi_j^*(y)\varphi_j(x)u(x,y)\,.$$

Defining

$$H(x,y) = \delta(x-y)\hat{T} + \delta(x-y)v_{\text{ext}}(y) + \delta(x-y)v_{\text{H}}(y) + \Sigma_{\text{x}}(x,y)\,,$$

the Hartree–Fock Hamiltonian $\hat{H}$ acts via the integral transform

$$\hat{H}\varphi(x) = \int H(x,y)\varphi(y)\,\mathrm{d}y\,.$$

When our one-dimensional space is discretized to points on a grid, $H(x,y)$ itself is a two-dimensional array, and acts as the Hamiltonian on the now one-dimensional arrays $\{\varphi_j\}_{j=1}^N$.

We solve the Hartree–Fock equation

$$\hat{H}\varphi_j = \varepsilon_j\varphi_j$$

to obtain a new set of orbitals and their corresponding eigenvalues.

Using these new orbitals the above process is iterated until a self-consistent solution is reached (i.e. until the change in the new electron density n is small enough compared to that of the previous iteration).

4.5.2 Time-dependence

A perturbing potential v_{ptrb} is added to the ground state system so that it now has Hamiltonian $\hat{T} + \hat{U} + \hat{V}_{\text{ext}} + \hat{V}_{\text{ptrb}}$.

We construct the Time-Dependent Hartree–Fock Hamiltonian, which again acts via the same integral transform as explained previously, but now with a perturbation potential term added as

$$H^{\text{TD}}(x,y) = H(x,y) + \delta(x-y)v_{\text{ptrb}}(y)\,.$$

Beginning with the ground state (initial time) orbitals and energies $\{\varphi_j(t=0), \varepsilon_j(t=0)\}_{j=1}^N$ as already calculated, we apply the Crank–Nicolson method to evolve the system forward in time by a step of size δt:

$$\{\varphi_j(t)\}_{j=1}^N \to \{\varphi_j(t = t + \delta t)\}_{j=1}^N\,.$$

This process is repeated until the desired duration of the time-evolution has been simulated. The electron density of the system is calculated from the orbitals at each time-step.

4.6 HYB (Hybrid)

The HYB code solves the Hybrid DFT equation (containing a linear combination of the LDA exchange-correlation potential and Fock operators) to approximately calculate the ground state density of a one-dimensional finite system of N like-spin electrons, with Hamiltonian $\hat{T} + \hat{U} + \hat{V}_{\text{ext}}$, where $\hat{U}$ is the operator for the softened Coulomb interaction potential given by

$$u(x,y) = (1 + |x-y|)^{-1}\,.$$

A perturbing potential may be applied to the ground state system, and the time-dependent Hybrid DFT equation solved approximately to calculate the system's time evolution.

4.6.1 Calculating the ground state

Orbitals and corresponding non-degenerate energy eigenvalues $\{\varphi_j, \varepsilon_j\}_{j=1}^N$ for a non-interacting system of N electrons are first calculated from the single-particle Schrödinger equation with Hamiltonian $\hat{T} + \hat{V}_{\text{ext}}$.

We then proceed to calculate the ground state electron density of this system:

$$n(x) = \sum_{j=1}^{N} |\varphi_j(x)|^2 \,.$$

From this we find the Hartree potential

$$v_{\text{H}}(x) = \int n(y) u(x,y) \, \mathrm{d}y \,,$$

the Fock matrix (the self-energy in the exchange-only approximation)

$$\Sigma_{\text{x}}(x,y) = -\sum_{j=1}^{N} \varphi_j^*(y)\varphi_j(x)u(x,y) \,,$$

and the LDA exchange-correlation potential $v_{\text{xc}}^{\text{LDA}}$ (see iDEA LDA).

Let $\alpha \in [0,1]$ be a fixed parameter which determines the linear mixing of Hartree-Fock and LDA. Then defining

$$H_\alpha(x,y) = \delta(x-y)\hat{T} + \delta(x-y)v_{\text{ext}}(y) + \delta(x-y)v_{\text{H}}(y) + \alpha\Sigma_{\text{x}}(x,y) + (1-\alpha)\delta(x-y)v_{\text{xc}}^{\text{LDA}}(y) \,,$$

the Hybrid Hamiltonian $\hat{H}_\alpha$ acts via the integral transform

$$\hat{H}_\alpha \varphi(x) = \int H_\alpha(x,y)\varphi(y) \, \mathrm{d}y \,.$$

When our one-dimensional space is discretized to points on a grid, $H_\alpha(x,y)$ itself is a two-dimensional array with a parameter α, and acts as the Hamiltonian on the now one-dimensional arrays $\{\varphi_j\}_{j=1}^N$.

We solve the Hybrid equation

$$\hat{H}_\alpha \varphi_j = \varepsilon_j \varphi_j$$

to obtain a new set of orbitals and their corresponding eigenvalues.

Using these new orbitals the above process is iterated until a self-consistent solution is reached (i.e. until the change in the new electron density n is small enough compared to that of the previous iteration).

The parameter α may be specified manually, however it is also possible to determine an "optimal" value for α by considering the generalized Koopmans' theorem, which gives conditions under which the model best describes a physical system.

4.6.2 Time-dependence

A perturbing potential v_{ptrb} is added to the ground state system so that it now has Hamiltonian $\hat{T} + \hat{U} + \hat{V}_{\text{ext}} + \hat{V}_{\text{ptrb}}$.

We construct the Time-Dependent Hybrid Hamiltonian, which again acts via the same integral transform as explained previously, but now with a perturbation potential term added as

$$H_\alpha^{\text{TD}}(x,y) = H_\alpha(x,y) + \delta(x-y)v_{\text{ptrb}}(y) \,.$$

Beginning with the ground state (initial time) orbitals and energies $\{\varphi_j(t=0), \varepsilon_j(t=0)\}_{j=1}^N$ as already calculated, we apply the Crank–Nicolson method to evolve the system forward in time by a step of size δt:

$$\{\varphi_j(t)\}_{j=1}^N \rightarrow \{\varphi_j(t = t + \delta t)\}_{j=1}^N \,.$$

This process is repeated until the desired duration of the time-evolution has been simulated. The electron density of the system is calculated from the orbitals at each time-step.

4.7 NON (non-interacting)

The NON code solves the single-particle time-independent Schrödinger equation (TISE) for one-dimensional finite systems of non-interacting electrons. A perturbing potential in then applied to the ground-state system and its evolution is calculated through solving the single-particle time-dependent Schrödinger equation (TDSE).

4.7.1 Calculating the ground-state

The Hamiltonian of a system of non-interacting electrons subject to a specified external potential is constructed, and from this a set of non-interacting orbitals are computed through solving the single-particle TISE:

$$\{\phi_j, \varepsilon_j\}.$$

From this, the ground-state density is calculated:

$$n(x) = \sum_{j \in \mathrm{occ}} |\phi_j|^2.$$

4.7.2 Time-dependence

After the ground-state is found, the perturbing potential is applied to the ground-state Hamiltonian, $\hat{H} = \hat{H}_0 + \delta\hat{V}_{\mathrm{ext}}$. The system's evolution is calculated by propagating the ground-state orbitals through real time using the Crank-Nicolson method.

CHAPTER 5

Developers

5.1 Coding practises

If you'd like to contribute your changes back to iDEA, please follow good coding practises.

5.1.1 Write unit tests

For any new feature you add, make sure to add a **unit test** that checks you feature is working as intended.

- Naming convention: `test_<your_module>.py`
- start by copying a simple example, e.g. `test_NON.py`
- make sure your test is quick, it should run *in the blink of an eye*

5.1.2 Checklist

Before making a pull request, go through the following checklist:

1. Check that you didn't break the existing unit tests:

```
# run this in the base directory
python -m unittest discover
```

2. Check that the documentation builds fine:

```
cd doc/
bash make_doc.sh
```

3. Check the test coverage for your module (should be close to 100%):

```
firefox doc/_build/coverage/index.html
```

5.2 Adding to iDEA

The iDEA source code is managed using the git version control system.

5.2.1 Committing changes locally

Before you start committing, make sure that your environment is properly configured:

```
git config --global user.name "John Smith"
git config --global user.email "john.smith@york.ac.uk"
```

Once you have made a set of changes you are happy with, you can commit the change set to your local repository. This will ensure that as you pull changes from the central repository they will be automatically integrated into your work. To see the list of files you have changed run

```
git status
```

Then to add a file you want to commit run

```
git add file_name
```

Once you have finished adding files you can commit your changes locally using

```
git commit
```

You will be prompted to enter a commit message to describe your changes and save the file. Your changes are now committed!

5.2.2 Contributing your changes to iDEA

Before contributing your changes back to iDEA, make sure to comply with our *best practises*.

1. Fork the iDEA repository on github
2. Assuming you've already committed your changes to a local repo, add a remote to your fork:

   ```
   git remote add fork https://github.com/[GitHub Username]/idea-public.git
   ```

3. Pull from your fork (to synchronize), and then push local changes back to github

   ```
   git pull fork master # may be asked to merge
   git push fork master
   ```

4. Create a pull request from your fork to the iDEA repo on github (you'll need to click on *compare across forks*)

Once a core developer maintainer has reviewed your pull request, your changes will be incorporated into iDEA.

5.2.3 Pulling the latest changes from iDEA

As the development of iDEA is progressing, you'll need to update your fork and your local repository from time to time.

Updating your local repository:

```
git remote add upstream git@github.com:godby-group/idea-public.git
git pull upstream master
```

After this, also update your fork on github:

```
git push fork master
```

Note: You will not be able to perfrom this pull if you have untracked changes, you should first commit your changes as described above. If you do not wish to commit the untracked changes and simply want to remove them run

```
git stash
```

5.3 Adding documentation

The documentation of iDEA consists of two main parts:

- the `doc/` folder, which contains documentation written in the intuitive reStructuredText (reST) format
- python docstrings in the iDEA source code that document the functionality of its modules, classes, functions, etc. (specifically, we follow the numpy convention)

Both are important and you are most welcome to contribute to either of them.

In order to generate a browseable HTML version of the documentation (which is displayed on the iDEA home page), we use a tool called Sphinx, which also takes care of producing a documentation of the source code (the API documentation).

Once you are done editing the documentation, produce an updated HTML version

```
cd doc
bash make_doc.sh
```

See *Generating the documentation* for details.

5.3.1 A short reST demo

- Write your mathematical formulae using LaTeX, in line $\exp(-i2\pi)$ or displayed

$$f(x) = \int_0^\infty \exp\left(\frac{x^2}{2}\right)\,dx$$

- You want to refer to a particular function or class? You can!

`iDEA.RE.`**`calculate_ground_state`**(*pm*, *approx*, *density_approx*, *v_ext*, *K*)

Calculates the exact ground-state Kohn-Sham potential by solving the ground-state Kohn-Sham equations and iteratively correcting v_ks. The exact ground-state Kohn-Sham eigenfunctions, eigenenergies and electron density are then calculated.

$$V_{\mathrm{KS}}(x) \rightarrow V_{\mathrm{KS}}(x) + \mu[n^p_{\mathrm{KS}}(x) - n^p_{\mathrm{approx}}(x)]$$

Parameters

pm [object] Parameters object

approx [string] The approximation used to calculate the electron density

density_approx [array_like] 1D array of the ground-state electron density from the approximation, indexed as density_approx[space_index]

v_ext [array_like] 1D array of the unperturbed external potential, indexed as v_ext[space_index]

K [array_like] 2D array of the kinetic energy matrix, index as K[band,space_index]

returns array_like and array_like and array_like and array_like and Boolean 1D array of the ground-state Kohn-Sham potential, indexed as v_ks[space_index]. 1D array of the ground-state Kohn-Sham electron density, indexed as density_ks[space_index]. 2D array of the ground-state Kohn-Sham eigenfunctions, indexed as wavefunctions_ks[space_index,eigenfunction]. 1D array containing the ground-state Kohn-Sham eigenenergies, indexed as energies_ks[eigenenergies]. Boolean - True if file containing exact Kohn-Sham potential is found, False if file is not found.

- Add images/plots to `iDEA/doc/_static` and then include them

- Check out the source of any page via the link in the bottom right corner.

reST source of this page:

```
********************
Adding documentation
********************

The documentation of iDEA consists of two main parts:

 * the :code:`doc/` folder, which contains documentation written in the intuitive
   `reStructuredText (reST) <http://sphinx-doc.org/rest.html#rst-primer>`_ format
 * `python docstrings <https://www.python.org/dev/peps/pep-0257/>`_ in the iDEA
   source code that document the functionality of its modules, classes, functions,
   etc. (specifically, we follow the
   `numpy convention <https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.
↪rst.txt>`_)

Both are important and you are most welcome to contribute to either of them.

In order to generate a browseable HTML version of the documentation (which is
displayed on the iDEA home page), we use a tool called
`Sphinx <http://sphinx-doc.org>`_, which also takes care of producing a
documentation of the source code (the API documentation).

Once you are done editing the documentation, produce an updated HTML version

.. code-block:: bash

    cd doc
```

(continues on next page)

(continued from previous page)

```
    bash make_doc.sh

See :ref:`generate-documentation` for details.


A short reST demo
------------------



  * Write your mathematical formulae using LaTeX,
    in line :math:`\exp(-i2\pi)` or displayed

    .. math:: f(x) = \int_0^\infty \exp\left(\frac{x^2}{2}\right)\,dx

  * You want to refer to a particular function or class? You can!

    .. autofunction:: iDEA.RE.calculate_ground_state
       :noindex:
  * Add images/plots to ``iDEA/doc/_static`` and then include them

    .. image:: ../_static/logo.png

  * Check out the source of any page via the link
    in the bottom right corner.

|

reST source of this page:

.. literalinclude:: documentation.rst
```

5.4 Changelog

- **v0.1.0** (2018-08-15)
  - Initial release of public version, relevant codes to be released were taken from private version 2.4.0.

5.5 Related Codes

Codes related to iDEA can be found in `iDEA/old`.

- PERiDEA

  An old version of iDEA (just EXT and RE) with periodic boundary conditions

CHAPTER 6

iDEA

6.1 iDEA package

6.1.1 Submodules

6.1.2 iDEA.EXT1 module

Calculates the exact ground-state electron density and energy for a system of one electron through solving the Schroedinger equation. If the system is perturbed, the time-dependent electron density and current density are calculated. Excited states of the unperturbed system can also be calculated.

`iDEA.EXT1.`**`calculate_current_density`**`(`*pm*, *density*`)`

Calculates the current density from the time-dependent electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index]

`iDEA.EXT1.`**`construct_A`**`(`*pm*, *H*`)`

Constructs the sparse matrix A, used when solving Ax=b in the Crank-Nicholson propagation.

$$A = I + i\frac{\delta t}{2}H$$

Parameters

pm [object] Parameters object

H [array_like] 2D array containing the band elements of the Hamiltonian matrix, indexed as H[band,space_index]

returns sparse_matrix The sparse matrix A

`iDEA.EXT1.`**`construct_K`**(*pm*)

Stores the band elements of the kinetic energy matrix in lower form. The kinetic energy matrix is constructed using a three-point, five-point or seven-point stencil. This yields an NxN band matrix (where N is the number of grid points). For example with N=6 and a three-point stencil:

$$K = -\frac{1}{2}\frac{d^2}{dx^2} = -\frac{1}{2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\frac{1}{\delta x^2} = [\frac{1}{\delta x^2}, -\frac{1}{2\delta x^2}]$$

Parameters

pm [object] Parameters object

returns array_like 2D array containing the band elements of the kinetic energy matrix, indexed as K[band,space_index]

`iDEA.EXT1.`**`main`**(*parameters*)

Calculates the ground-state of the system. If the system is perturbed, the time evolution of the perturbed system is then calculated.

Parameters

parameters [object] Parameters object

returns object Results object

6.1.3 iDEA.EXT2 module

Calculates the exact ground-state electron density and energy for a system of two interacting electrons through solving the many-electron Schroedinger equation. If the system is perturbed, the time-dependent electron density and current density are calculated.

`iDEA.EXT2.`**`calculate_current_density`**(*pm*, *density*)

Calculates the current density from the time-dependent electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index]

`iDEA.EXT2.`**`calculate_density`**(*pm*, *wavefunction_2D*)

Calculates the electron density from the two-electron wavefunction.

$$n(x) = 2\int_{-x_{\max}}^{x_{\max}} |\Psi(x, x_2)|^2 dx_2$$

Parameters

pm [object] Parameters object

wavefunction [array_like] 2D array of the wavefunction, indexed as wavefunction_2D[space_index_1,space_index_2]

returns array_like 1D array of the density, indexed as density[space_index]

iDEA.EXT2.**calculate_energy**(*pm*, *wavefunction_reduced*, *wavefunction_reduced_old*)

Calculates the energy of the system.

$$E = -\ln\left(\frac{|\Psi(x_1, x_2, \tau)|}{|\Psi(x_1, x_2, \tau - \delta\tau)|}\right)\frac{1}{\delta\tau}$$

Parameters

pm [object] Parameters object

wavefunction_reduced [array_like] 1D array of the reduced wavefunction at t, indexed as wavefunction_reduced[space_index_1_2]

wavefunction_reduced_old [array_like] 1D array of the reduced wavefunction at t-dt, indexed as wavefunction_reduced_old[space_index_1_2]

returns float Energy of the system

iDEA.EXT2.**construct_A_reduced**(*pm*, *reduction_matrix*, *expansion_matrix*, *td*)

Constructs the reduced form of the sparse matrix A.

$$\text{Imaginary time} : A = I + \frac{\delta\tau}{2}H$$
$$\text{Real time} : A = I + i\frac{\delta t}{2}H$$

$$A_{\text{red}} = RAE$$

where R = reduction matrix and E = expansion matrix

Parameters

pm [object] Parameters object

reduction_matrix [sparse_matrix] Sparse matrix used to reduce the wavefunction (remove indistinct elements) by exploiting the exchange antisymmetry

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

td [integer] 0 for imaginary time, 1 for real time

returns sparse_matrix Reduced form of the sparse matrix A, used when solving the equation Ax=b

iDEA.EXT2.**construct_antisymmetry_matrices**(*pm*)

Constructs the reduction and expansion matrices that are used to exploit the exchange antisymmetry of the wavefunction.

$$\Psi(x_1, x_2) = -\Psi(x_2, x_1)$$

Parameters

pm [object] Parameters object

returns sparse_matrix and sparse_matrix Reduction matrix used to reduce the wavefunction (remove indistinct elements). Expansion matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction.

`iDEA.EXT2.`**`initial_wavefunction`**(*pm*)

Generates the initial condition for the Crank-Nicholson imaginary time propagation.

$$\Psi(x_1, x_2) = \frac{1}{\sqrt{2}}\big(\phi_1(x_1)\phi_2(x_2) - \phi_2(x_1)\phi_1(x_2)\big)$$

Parameters

pm [object] Parameters object

returns array_like 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2]

`iDEA.EXT2.`**`main`**(*parameters*)

Calculates the ground-state of the system. If the system is perturbed, the time evolution of the perturbed system is then calculated.

Parameters

parameters [object] Parameters object

returns object Results object

`iDEA.EXT2.`**`non_approx`**(*pm*)

Calculates the two lowest non-interacting eigenstates of the system. These can then be expressed in Slater determinant form as an approximation to the exact many-electron wavefunction.

$$\bigg(-\frac{1}{2}\frac{d^2}{dx^2} + V_{\text{ext}}(x)\bigg)\phi_j(x) = \varepsilon_j\phi_j(x)$$

Parameters

pm [object] Parameters object

returns array_like and array_like 1D array of the 1st non-interacting eigenstate, indexed as eigenstate_1[space_index]. 1D array of the 2nd non-interacting eigenstate, indexed as eigenstate_2[space_index].

`iDEA.EXT2.`**`qho_approx`**(*pm, n*)

Calculates the nth eigenstate of the quantum harmonic oscillator, and shifts to ensure it is neither an odd nor an even function (necessary for the Gram-Schmidt algorithm).

$$\bigg(-\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}\omega^2 x^2\bigg)\phi_n(x) = \varepsilon_n\phi_n(x)$$

$$\phi_n(x) = \frac{1}{\sqrt{2^n n!}}\bigg(\frac{\omega}{\pi}\bigg)^{1/4} e^{-\frac{\omega x^2}{2}} H_n\bigg(\sqrt{\omega}x\bigg)$$

Parameters

pm [object] Parameters object

n [integer] Principle quantum number

returns array_like 1D array of the nth eigenstate, indexed as eigenstate[space_index]

`iDEA.EXT2.`**`solve_imaginary_time`**(*pm, A_reduced, C_reduced, wavefunction_reduced, expansion_matrix*)

Propagates the initial wavefunction through imaginary time using the Crank-Nicholson method to find the ground-state of the system.

$$\Big(I + \frac{\delta\tau}{2}H\Big)\Psi(x_1, x_2, \tau + \delta\tau) = \Big(I - \frac{\delta\tau}{2}H\Big)\Psi(x_1, x_2, \tau)$$
$$\Psi(x_1, x_2, \tau) = \sum_m c_m e^{-\varepsilon_m \tau}\phi_m \implies \lim_{\tau\to\infty}\Psi(x_1, x_2, \tau) = \phi_0$$

Parameters

pm [object] Parameters object

A_reduced [sparse_matrix] Reduced form of the sparse matrix A, used when solving the equation Ax=b

C_reduced [sparse_matrix] Reduced form of the sparse matrix C, defined as C=-A+2I

wavefunction_reduced [array_like] 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2]

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

returns float and array_like Energy of the ground-state system. 1D array of the ground-state wavefunction, indexed as wavefunction[space_index_1_2].

`iDEA.EXT2.`**`solve_real_time`**(*pm*, *A_reduced*, *C_reduced*, *wavefunction*, *reduction_matrix*, *expansion_matrix*)

Propagates the ground-state wavefunction through real time using the Crank-Nicholson method to find the time-evolution of the perturbed system.

$$\Big(I + i\frac{\delta t}{2}H\Big)\Psi(x_1, x_2, t + \delta t) = \Big(I - i\frac{\delta t}{2}H\Big)\Psi(x_1, x_2, t)$$

Parameters

pm [object] Parameters object

A_reduced [sparse_matrix] Reduced form of the sparse matrix A, used when solving the equation Ax=b

C_reduced [sparse_matrix] Reduced form of the sparse matrix C, defined as C=-A+2I

wavefunction [array_like] 1D array of the ground-state wavefunction, indexed as wavefunction[space_index_1_2]

reduction_matrix [sparse_matrix] Sparse matrix used to reduce the wavefunction (remove indistinct elements) by exploiting the exchange antisymmetry

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

returns array_like and array_like 2D array of the time-dependent density, indexed as density[time_index,space_index]. 2D array of the current density, indexed as current_density[time_index,space_index].

6.1.4 iDEA.EXT3 module

Calculates the exact ground-state electron density and energy for a system of three interacting electrons through solving the many-electron Schroedinger equation. If the system is perturbed, the time-dependent electron density and current density are calculated.

`iDEA.EXT3.`**`calculate_current_density`**(*pm*, *density*)

Calculates the current density from the time-dependent electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index]

`iDEA.EXT3.`**`calculate_density`**(*pm*, *wavefunction_3D*)

Calculates the electron density from the three-electron wavefunction.

$$n(x) = 3\int_{-x_{\max}}^{x_{\max}} |\Psi(x, x_2, x_3)|^2 dx_2\, dx_3$$

Parameters

pm [object] Parameters object

wavefunction [array_like] 3D array of the wavefunction, indexed as wavefunction_3D[space_index_1,space_index_2,space_index_3]

returns array_like 1D array of the density, indexed as density[space_index]

`iDEA.EXT3.`**`calculate_energy`**(*pm*, *wavefunction_reduced*, *wavefunction_reduced_old*)

Calculates the energy of the system.

$$E = -\ln\left(\frac{|\Psi(x_1, x_2, x_3, \tau)|}{|\Psi(x_1, x_2, x_3, \tau - \delta\tau)|}\right)\frac{1}{\delta\tau}$$

Parameters

pm [object] Parameters object

wavefunction_reduced [array_like] 1D array of the reduced wavefunction at t, indexed as wavefunction_reduced[space_index_1_2_3]

wavefunction_reduced_old [array_like] 1D array of the reduced wavefunction at t-dt, indexed as wavefunction_reduced_old[space_index_1_2_3]

returns float Energy of the system

`iDEA.EXT3.`**`construct_A_reduced`**(*pm*, *reduction_matrix*, *expansion_matrix*, *td*)

Constructs the reduced form of the sparse matrix A.

$$\text{Imaginary time}: A = I + \frac{\delta\tau}{2}H$$
$$\text{Real time}: A = I + i\frac{\delta t}{2}H$$

$$A_{\text{red}} = RAE$$

where $R =$ reduction matrix and $E =$ expansion matrix

Parameters

pm [object] Parameters object

reduction_matrix [sparse_matrix] Sparse matrix used to reduce the wavefunction (remove indistinct elements) by exploiting the exchange antisymmetry

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

td [integer] 0 for imaginary time, 1 for real time

returns sparse_matrix Reduced form of the sparse matrix A, used when solving the equation Ax=b

iDEA.EXT3.**construct_antisymmetry_matrices**(*pm*)

Constructs the reduction and expansion matrices that are used to exploit the exchange antisymmetry of the wavefunction.

$$\Psi(x_1, x_2, x_3) = -\Psi(x_2, x_1, x_3) = \Psi(x_2, x_3, x_1)$$

Parameters

pm [object] Parameters object

returns sparse_matrix and sparse_matrix Reduction matrix used to reduce the wavefunction (remove indistinct elements). Expansion matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction.

iDEA.EXT3.**initial_wavefunction**(*pm*)

Generates the initial wavefunction for the Crank-Nicholson imaginary time propagation.

$$\Psi(x_1, x_2, x_3) = \frac{1}{\sqrt{6}}\big(\phi_1(x_1)\phi_2(x_2)\phi_3(x_3) - \phi_1(x_1)\phi_3(x_2)\phi_2(x_3) + \phi_3(x_1)\phi_1(x_2)\phi_2(x_3) - \phi_3(x_1)\phi_2(x_2)\phi_1(x_3) + \phi_2(x_1)\phi_3$$

Parameters

pm [object] Parameters object

returns array_like 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2_3]

iDEA.EXT3.**main**(*parameters*)

Calculates the ground-state of the system. If the system is perturbed, the time evolution of the perturbed system is then calculated.

Parameters

parameters [object] Parameters object

returns object Results object

iDEA.EXT3.**non_approx**(*pm*)

Calculates the three lowest non-interacting eigenstates of the system. These can then be expressed in Slater determinant form as an approximation to the exact many-electron wavefunction.

$$\bigg(-\frac{1}{2}\frac{d^2}{dx^2} + V_{\text{ext}}(x)\bigg)\phi_j(x) = \varepsilon_j\phi_j(x)$$

Parameters

pm [object] Parameters object

returns array_like and array_like and array_like 1D array of the 1st non-interacting eigenstate, indexed as eigenstate_1[space_index]. 1D array of the 2nd non-interacting eigenstate, indexed as eigenstate_2[space_index]. 1D array of the 3rd non-interacting eigenstate, indexed as eigenstate_3[space_index].

`iDEA.EXT3.`**`qho_approx`**(*pm*, *n*)

Calculates the nth eigenstate of the quantum harmonic oscillator, and shifts to ensure it is neither an odd nor an even function (necessary for the Gram-Schmidt algorithm).

$$\Big(-\frac{1}{2}\frac{d^2}{dx^2}+\frac{1}{2}\omega^2x^2\Big)\phi_n(x)=\varepsilon_n\phi_n(x)$$
$$\phi_n(x)=\frac{1}{\sqrt{2^n n!}}\Big(\frac{\omega}{\pi}\Big)^{1/4}e^{-\frac{\omega x^2}{2}}H_n\Big(\sqrt{\omega}x\Big)$$

Parameters

pm [object] Parameters object

n [integer] Principle quantum number

returns array_like 1D array of the nth eigenstate, indexed as eigenstate[space_index]

`iDEA.EXT3.`**`solve_imaginary_time`**(*pm*, *A_reduced*, *C_reduced*, *wavefunction_reduced*, *expansion_matrix*)

Propagates the initial wavefunction through imaginary time using the Crank-Nicholson method to find the ground-state of the system.

$$\Big(I+\frac{\delta\tau}{2}H\Big)\Psi(x_1,x_2,x_3,\tau+\delta\tau)=\Big(I-\frac{\delta\tau}{2}H\Big)\Psi(x_1,x_2,x_3,\tau)$$
$$\Psi(x_1,x_2,x_3,\tau)=\sum_m c_m e^{-\varepsilon_m\tau}\phi_m \implies \lim_{\tau\to\infty}\Psi(x_1,x_2,x_3,\tau)=\phi_0$$

Parameters

pm [object] Parameters object

A_reduced [sparse_matrix] Reduced form of the sparse matrix A, used when solving the equation Ax=b

C_reduced [sparse_matrix] Reduced form of the sparse matrix C, defined as C=-A+2I

wavefunction_reduced [array_like] 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2_3]

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

returns float and array_like Energy of the ground-state system. 1D array of the ground-state wavefunction, indexed as wavefunction[space_index_1_2_3].

`iDEA.EXT3.`**`solve_real_time`**(*pm*, *A_reduced*, *C_reduced*, *wavefunction*, *reduction_matrix*, *expansion_matrix*)

Propagates the ground-state wavefunction through real time using the Crank-Nicholson method to find the time-evolution of the perturbed system.

$$\Big(I+i\frac{\delta t}{2}H\Big)\Psi(x_1,x_2,x_3,t+\delta t)=\Big(I-i\frac{\delta t}{2}H\Big)\Psi(x_1,x_2,x_3,t)$$

Parameters

pm [object] Parameters object

A_reduced [sparse_matrix] Reduced form of the sparse matrix A, used when solving the equation Ax=b

C_reduced [sparse_matrix] Reduced form of the sparse matrix C, defined as C=-A+2I

wavefunction [array_like] 1D array of the ground-state wavefunction, indexed as wavefunction[space_index_1_2_3]

reduction_matrix [sparse_matrix] Sparse matrix used to reduce the wavefunction (remove indistinct elements) by exploiting the exchange antisymmetry

expansion_matrix [sparse_matrix] Sparse matrix used to expand the reduced wavefunction (insert indistinct elements) to get back the full wavefunction

returns array_like and array_like 2D array of the time-dependent density, indexed as density[time_index,space_index]. 2D array of the current density, indexed as current_density[time_index,space_index].

6.1.5 iDEA.EXT_cython module

Contains the cython modules that are called within EXT2 and EXT3. Cython is used for operations that are very expensive to do in Python, and performance speeds are close to C.

iDEA.EXT_cython.**continuity_eqn**()

Calculates the electron current density of the system at a particular time step by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density_new [array_like] 1D array of the electron density at time t

density_old [array_like] 1D array of the electron density at time t-dt

returns array_like 1D array of the electron current density at time t

iDEA.EXT_cython.**expansion_three**()

Calculates the coordinates and data of the non-zero elements of the three-electron expansion matrix that is used to exploit the exchange antisymmetry of the three-electron wavefunction.

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the expansion matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the expansion matrix

coo_data [array_like] 1D array of the non-zero elements of the expansion matrix

returns array_like and array_like and array_like Populated 1D COOrdinate holding arrays and 1D data holding array for the non-zero elements of the expansion matrix

iDEA.EXT_cython.**expansion_two**()

Calculates the coordinates and data of the non-zero elements of the two-electron expansion matrix that is used to exploit the exchange antisymmetry of the two-electron wavefunction.

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the expansion matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the expansion matrix

coo_data [array_like] 1D array of the non-zero elements of the expansion matrix

returns array_like and array_like and array_like Populated 1D COOrdinate holding arrays and 1D data holding array for the non-zero elements of the expansion matrix

`iDEA.EXT_cython.`**`hamiltonian_three`**`()`

Calculates the coordinates and data of the non-zero elements of the three-electron Hamiltonian matrix.

$$\hat{H} = \sum_{i=1}^{3} \hat{K}_i + \sum_{i=1}^{3} \hat{V}_{\text{ext}}(x_i) + \sum_{i=1}^{3} \sum_{j>i}^{3} \hat{V}_{\text{int}}(x_i, x_j)$$

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the Hamiltonian matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the Hamiltonian matrix

coo_data [array_like] 1D array of the non-zero elements of the Hamiltonian matrix

td [integer] 0 for imaginary time, 1 for real time

returns array_like and array_like and array_like Populated 1D COOrdinate holding arrays and 1D data holding array for the non-zero elements of the Hamiltonian matrix

`iDEA.EXT_cython.`**`hamiltonian_two`**`()`

Calculates the coordinates and data of the non-zero elements of the two-electron Hamiltonian matrix.

$$\hat{H} = \sum_{i=1}^{2} \hat{K}_i + \sum_{i=1}^{2} \hat{V}_{\text{ext}}(x_i) + \sum_{i=1}^{2} \sum_{j>i}^{2} \hat{V}_{\text{int}}(x_i, x_j)$$

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the Hamiltonian matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the Hamiltonian matrix

coo_data [array_like] 1D array of the non-zero elements of the Hamiltonian matrix

td [integer] 0 for imaginary time, 1 for real time

returns array_like and array_like and array_like Populated 1D COOrdinate holding arrays and 1D data holding array for the non-zero elements of the Hamiltonian matrix

`iDEA.EXT_cython.`**`imag_pot_three`**`()`

Calculates the imaginary component of the perturbing potential to be added to the main diagonal of the three-electron Hamiltonian matrix if imaginary potentials are used.

Parameters

pm [object] Parameters object

returns array_like 1D array of the perturbing potential

`iDEA.EXT_cython.`**`imag_pot_two`**`()`

Calculates the imaginary component of the perturbing potential to be added to the main diagonal of the two-electron Hamiltonian matrix if imaginary potentials are used.

Parameters

pm [object] Parameters object

returns array_like 1D array of the perturbing potential

`iDEA.EXT_cython.`**`reduction_three`**`()`

Calculates the coordinates and data of the non-zero elements of the three-electron reduction matrix that is used to exploit the exchange antisymmetry of the three-electron wavefunction.

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the reduction matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the reduction matrix

returns array_like and array_like Populated 1D COOrdinate holding arrays for the non-zero elements of the reduction matrix

`iDEA.EXT_cython.`**`reduction_two`**`()`

Calculates the coordinates and data of the non-zero elements of the two-electron reduction matrix that is used to exploit the exchange antisymmetry of the two-electron wavefunction.

Parameters

pm [object] Parameters object

coo_1 [array_like] 1D COOrdinate holding array for the non-zero elements of the reduction matrix

coo_2 [array_like] 1D COOrdinate holding array for the non-zero elements of the reduction matrix

returns array_like and array_like Populated 1D COOrdinate holding arrays for the non-zero elements of the reduction matrix

`iDEA.EXT_cython.`**`single_index_three`**`()`

Takes every permutation of the three electron indices and creates a single unique index.

Parameters

j [integer] First electron index

k [integer] Second electron index

l [integer] Third electron index

grid [integer] Number of spatial grid points in the system

returns integer Single unique index

`iDEA.EXT_cython.`**`single_index_two`**`()`

Takes every permutation of the two electron indices and creates a single unique index.

Parameters

j [integer] First electron index

k [integer] Second electron index

grid [integer] Number of spatial grid points in the system

returns integer Single unique index

iDEA.EXT_cython.**wavefunction_three**()

Constructs the initial three-electron wavefunction in reduced form from three single-electron eigenstates.

$$\Psi(x_1,x_2,x_3) = \frac{1}{\sqrt{6}}\big(\phi_1(x_1)\phi_2(x_2)\phi_3(x_3) - \phi_1(x_1)\phi_3(x_2)\phi_2(x_3) + \phi_3(x_1)\phi_1(x_2)\phi_2(x_3) - \phi_3(x_1)\phi_2(x_2)\phi_1(x_3) + \phi_2(x_1)\phi_3$$

Parameters

pm [object] Parameters object

eigenstate_1 [array_like] 1D array of the 1st single-electron eigenstate, indexed as eigenstate_1[space_index]

eigenstate_2 [array_like] 1D array of the 2nd single-electron eigenstate, indexed as eigenstate_2[space_index]

eigenstate_3 [array_like] 1D array of the 3rd single-electron eigenstate, indexed as eigenstate_3[space_index]

returns array_like 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2_3]

iDEA.EXT_cython.**wavefunction_two**()

Constructs the two-electron initial wavefunction in reduced form from two single-electron eigenstates.

$$\Psi(x_1,x_2) = \frac{1}{\sqrt{2}}\big(\phi_1(x_1)\phi_2(x_2) - \phi_2(x_1)\phi_1(x_2)\big)$$

Parameters

pm [object] Parameters object

eigenstate_1 [array_like] 1D array of the 1st single-electron eigenstate, indexed as eigenstate_1[space_index]

eigenstate_2 [array_like] 1D array of the 2nd single-electron eigenstate, indexed as eigenstate_2[space_index]

returns array_like 1D array of the reduced wavefunction, indexed as wavefunction_reduced[space_index_1_2]

6.1.6 iDEA.HF module

Computes ground-state and time-dependent charge density in the Hartree-Fock approximation. The code outputs the ground-state charge density, the energy of the system and the Hartree-Fock orbitals.

iDEA.HF.**calculate_current_density**(*pm*, *density*)

Calculates the current density from the time-dependent (and ground-state) electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \nabla \cdot j = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index] `iDEA.HF.`**`crank_nicolson_step`**(*pm*, *waves*, *H*)

Solves Crank Nicolson Equation

$$\left(\nVdash + i\frac{dt}{2}H\right)\Psi(x,t+dt) = \left(\nVdash - i\frac{dt}{2}H\right)\Psi(x,t)$$

for $\Psi(x,t+dt)$.

Parameters

pm [object] Parameters object

total_td_density [array_like] Time dependent density of the system indexed as total_td_density[time_index][space_index]

returns array_like Time dependent current density indexed as current_density[time_index][space_index]

`iDEA.HF.`**`electron_density`**(*pm*, *orbitals*)

Compute density for given orbitals

Parameters

pm [object] Parameters object

orbitals: array_like Array of properly normalised orbitals

Returns

n: array_like electron density

`iDEA.HF.`**`fock`**(*pm*, *eigf*)

Constructs Fock operator from a set of orbitals

$$\Sigma_{\text{x}}(x,y) = -\sum_{j=1}^{N}\varphi_j^*(y)\varphi_j(x)u(x,y)$$

where u(x,y) denotes the appropriate Coulomb interaction.

Parameters

pm [object] Parameters object

eigf [array_like] Eigenfunction orbitals

Returns

F: array_like Fock matrix

`iDEA.HF.`**`groundstate`**(*pm*, *H*)

Diagonalises Hamiltonian H

$$\hat{H}\varphi_j = \varepsilon_j\varphi_j$$

Parameters

pm: object Parameters object

H: array_like Hamiltonian matrix (band form)

Returns

n: array_like density

eigf: array_like normalised orbitals, index as eigf[space_index,orbital_number]

eigv: array_like orbital energies

iDEA.HF.**hamiltonian**(*pm*, *wfs*, *perturb=False*)

Compute HF Hamiltonian

Computes HF Hamiltonian from a given set of single-particle states

Parameters

pm [object] Parameters object

wfs array_like single-particle states

perturb: bool If True, add perturbation to external potential (for time-dep. runs)

returns array_like Hamiltonian matrix

iDEA.HF.**hartree**(*pm*, *density*)

Computes Hartree potential for a given density

$$v_{\mathrm{H}}(x) = \int n(y)u(x,y)\,\mathrm{d}y$$

Parameters

pm [object] Parameters object

density [array_like] given density

returns array_like Hartree potential

iDEA.HF.**main**(*parameters*)

Performs Hartree-fock calculation

Parameters

parameters [object] Parameters object

returns object Results object

iDEA.HF.**total_energy**(*pm*, *eigf*, *eigv*)

Calculates total energy of Hartree-Fock wave function

Parameters

pm [array_like] external potential

eigf [array_like] eigenfunctions

eigv [array_like] eigenvalues

returns float

6.1.7 iDEA.HYB module

Computes ground-state and time-dependent charge density in the Hybrid Hartree-Fock-LDA approximation.

The code outputs the ground-state charge density, the energy of the system and the single-quasiparticle orbitals.

iDEA.HYB.**calc_with_alpha**(*pm*, *alpha*, *occupations*)

Calculate with given alpha.

Perform hybrid calculation with given alpha.

Parameters

alpha float HF-LDA mixing parameter (1 = all HF, 0 = all LDA)

occupations: array_like orbital occupations

returns density, eigf, eigv, E Hybrid Density, orbitals and total energy

`iDEA.HYB.`**`fractional_run`**(*pm*, *results*, *occupations*, *fractions*)

`iDEA.HYB.`**`hamiltonian`**(*pm*, *eigf*, *density*, *alpha*, *occupations*, *perturb=False*)

Compute HF Hamiltonian.

Computes HYB Hamiltonian from a given set of single-particle states.

$$H_\alpha(x,y) = \delta(x-y)\hat{T} + \delta(x-y)v_{\text{ext}}(y) + \delta(x-y)v_{\text{H}}(y) + \alpha\Sigma_{\text{x}}(x,y) + (1-\alpha)\delta(x-y)v_{\text{xc}}^{\text{LDA}}(y)$$

Parameters

eigf array_like single-particle states

density array_like electron density

alpha float HF-LDA mixing parameter (1 = all HF, 0 = all LDA)

occupations: array_like orbital occupations

perturb: bool If True, add perturbation to external potential (for time-dep. runs)

returns array_like Hamiltonian matrix

`iDEA.HYB.`**`main`**(*parameters*)

Performs Hybrid calculation.

Parameters

parameters [object] Parameters object

returns object Results object

`iDEA.HYB.`**`n_minus_one_run`**(*pm*, *results*, *alphas*, *occupations*)

Calculate for $N-1$ electron run.

Calculate total energy and LUMO eigenvalue of $N-1$ electron system.

Parameters

results Results Object object to add results to

alphas: array_like range of alphas to use

occupations: array_like orbital occupations

`iDEA.HYB.`**`n_run`**(*pm*, *results*, *alphas*, *occupations*)

Calculate for N electron run.

Calculate total energy and HOMO eigenvalue of N electron system.

Parameters

results Results Object object to add results to

alphas: array_like range of alphas to use

occupations: array_like orbital occupations

`iDEA.HYB.`**`optimal_alpha`**(*pm*, *results*, *alphas*, *occupations*)

Calculate optimal alpha.

Calculate over range of alphas to determine optimal alpha.

Parameters

results Results Object object to add results to

alphas: array_like range of alphas to use

occupations: array_like orbital occupations

`iDEA.HYB.`**`save_results`**(*pm*, *results*, *density*, *E*, *eigf*, *eigv*, *alpha*)

Saves hybrid results to outputs directory

6.1.8 iDEA.LDA module

Uses the [adiabatic] local density approximation ([A]LDA) to calculate the [time-dependent] electron density [and current] for a system of N electrons.

Computes approximations to V_KS, V_H, V_xc using the LDA self-consistently. For ground state calculations the code outputs the LDA orbitals and energies of the system, the ground-state charge density and Kohn-Sham potential. For time dependent calculations the code also outputs the time-dependent charge and current densities and the time-dependent Kohn-Sham potential.

Note: Uses the LDAs developed in *[Entwistle2018]* from finite slab systems and the HEG, in one dimension.

`iDEA.LDA.`**`DXC`**(*pm*, *n*)

Calculates the derivative of the exchange-correlation potential, necessary for the RPA preconditioner.

Parameters

pm [object] Parameters object

n [array_like] 1D array of the electron density, indexed as n[space_index]

returns array_like 1D array of the derivative of the exchange-correlation potential, indexed as D_xc[space_index]

`iDEA.LDA.`**`banded_to_full`**(*pm*, *H*)

Converts the Hamiltonian matrix in band form to the full matrix.

Parameters

pm [object] Parameters object

H [array_like] 2D array of the Hamiltonian matrix in band form, indexed as H[band,space_index]

Returns

H_full [array_like] 2D array of the Hamiltonian matrix in full form, indexed as H_full[space_index,space_index]

`iDEA.LDA.`**`calculate_current_density`**(*pm*, *density*)

Calculates the current density from the time-dependent electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index] `iDEA.LDA.`**`crank_nicolson_step`**(*pm*, *orbitals*, *H_full*)

Solves Crank Nicolson Equation

$$\left(I + i\frac{dt}{2}H\right)\Psi(x,t+dt) = \left(I - i\frac{dt}{2}H\right)\Psi(x,t)$$

Parameters

pm [object] Parameters object

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

H_full [array_like] 2D array of the Hamiltonian matrix in full form, indexed as H_full[space_index,space_index]

returns

`iDEA.LDA.`**`electron_density`**(*pm*, *orbitals*)

Calculates the electron density from the set of orbitals.

$$n(x) = \sum_{j=1}^{N} |\phi_j(x)|^2$$

Parameters

pm [object] Parameters object

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

Returns

density [array_like] 1D array of the electron density, indexed as density[space_index]

`iDEA.LDA.`**`groundstate`**(*pm*, *H*)

Calculates the ground-state of the system for a given potential.

$$\hat{H}\phi_j = \varepsilon_j \phi_j$$

Parameters

pm [object] Parameters object

H [array_like] 2D array of the Hamiltonian matrix in band form, indexed as H[band,space_index]

Returns

density [array_like] 1D array of the electron density, indexed as density[space_index]

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

eigenvalues [array_like] 1D array of the Kohn-Sham eigenvalues, indexed as eigenvalues[orbital_number]

`iDEA.LDA.`**`hamiltonian`**(*pm*, *v_ks=None*, *orbitals=None*, *perturbation=False*)

Constructs the Hamiltonian matrix in band form for a given Kohn-Sham potential.

$$\hat{H} = \hat{K} + \hat{V}_{\mathrm{KS}}$$

Parameters

pm [object] Parameters object

v_ks [array_like] 1D array of the Kohn-Sham potential, indexed as v_ks[space_index]

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

perturbation: bool

- True: Perturbed external potential
- False: Unperturbed external potential

Returns

H [array_like] 2D array of the Hamiltonian matrix in band form, indexed as H[band,space_index]

`iDEA.LDA.`**`hartree_energy`**`(`*pm*, *v_h*, *density*`)`

Calculates the Hartree energy of the ground-state system.

$$E_{\mathrm{H}}[n] = \frac{1}{2}\int\int U(x,x')n(x)n(x')dxdx' = \frac{1}{2}\int V_{\mathrm{H}}(x)n(x)dx$$

Parameters

pm [object] Parameters object

v_h [array_like] 1D array of the ground-state Hartree potential, indexed as v_h[space_index]

density [array_like] 1D array of the ground-state electron density, indexed as density[space_index]

returns float The Hartree energy of the ground-state system

`iDEA.LDA.`**`hartree_potential`**`(`*pm*, *density*`)`

Calculates the Hartree potential for a given electron density.

$$V_{\mathrm{H}}(x) = \int U(x,x')n(x')dx'$$

Parameters

pm [object] Parameters object

density [array_like] 1D array of the electron density, indexed as density[space_index]

returns array_like 1D array of the Hartree potential, indexed as v_h[space_index]

`iDEA.LDA.`**`kinetic`**`(`*pm*`)`

Stores the band elements of the kinetic energy matrix in lower form. The kinetic energy matrix is constructed using a three-point, five-point or seven-point stencil. This yields an NxN band matrix (where N is the number of grid points). For example with N=6 and a three-point stencil:

$$K = -\frac{1}{2}\frac{d^2}{dx^2} = -\frac{1}{2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\frac{1}{\delta x^2} = [\frac{1}{\delta x^2}, -\frac{1}{2\delta x^2}]$$

Parameters

pm [object] Parameters object

returns array_like 2D array containing the band elements of the kinetic energy matrix, indexed as K[band,space_index]

`iDEA.LDA.`**`kinetic_energy`**(*pm*, *orbitals*)

Calculates the kinetic energy from the Kohn-Sham orbitals.

$$T_s[n] = \sum_{j=1}^{N} \langle \phi_j | K | \phi_j \rangle$$

Parameters

pm [object] Parameters object

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

`iDEA.LDA.`**`ks_potential`**(*pm*, *density*, *perturbation=False*)

Calculates the Kohn-Sham potential from the electron density.

$$V_{\mathrm{KS}} = V_{\mathrm{ext}} + V_{\mathrm{H}} + V_{\mathrm{xc}}$$

Parameters

pm [object] Parameters object

density [array_like] 1D array of the electron density, indexed as density[space_index]

perturbation: bool

- True: Perturbed external potential
- False: Unperturbed external potential

Returns

v_ks [array_like] 1D array of the Kohn-Sham potential, indexed as v_ks[space_index]

`iDEA.LDA.`**`main`**(*parameters*)

Performs LDA calculation

Parameters

parameters [object] Parameters object

returns object Results object

`iDEA.LDA.`**`total_energy_eigf`**(*pm*, *orbitals*, *density=None*, *v_h=None*)

Calculates the total energy from the Kohn-Sham orbitals.

$$E[n] = \sum_{j=1}^{N} \langle \phi_j | K | \phi_j \rangle + E_H[n] + E_{xc}[n] + \int n(x) V_{\mathrm{ext}}(x) dx$$

Parameters

pm [object] Parameters object

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

density [array_like] 1D array of the electron density, indexed as density[space_index]

v_h [array_like] 1D array of the Hartree potential, indexed as v_h[space_index]

returns float Total energy

iDEA.LDA.**total_energy_eigv**(*pm*, *eigenvalues*, *orbitals=None*, *density=None*, *v_h=None*, *v_xc=None*)

Calculates the total energy from the Kohn-Sham eigenvalues.

$$E[n] = \sum_{j=1}^{N} \varepsilon_j + E_{xc}[n] - E_H[n] - \int n(x)V_{xc}(x)dx$$

Parameters

pm [object] Parameters object

eigenvalues [array_like] 1D array of the Kohn-Sham eigenvalues, indexed as eigenvalues[orbital_number]

orbitals [array_like] 2D array of the Kohn-Sham orbitals, index as orbitals[space_index,orbital_number]

density [array_like] 1D array of the electron density, indexed as density[space_index]

v_h [array_like] 1D array of the Hartree potential, indexed as v_h[space_index]

v_xc [array_like] 1D array of the exchange-correlation potential, indexed as v_xc[space_index]

returns float Total energy

iDEA.LDA.**xc_energy**(*pm*, *n*, *separate=False*)

LDA approximation for the exchange-correlation energy. Uses the LDAs developed in [Entwistle et al. 2018] from finite slab systems and the HEG.

$$E_{\mathrm{xc}}^{\mathrm{LDA}}[n] = \int \varepsilon_{\mathrm{xc}}(n)n(x)dx$$

Parameters

pm [object] Parameters object

n [array_like] 1D array of the electron density, indexed as n[space_index]

separate: bool

- True: Split the HEG exchange-correlation energy into separate exchange and correlation terms
- False: Just return the exchange-correlation energy

returns float Exchange-correlation energy

iDEA.LDA.**xc_potential**(*pm*, *n*, *separate=False*)

LDA approximation for the exchange-correlation potential. Uses the LDAs developed in [Entwistle et al. 2018] from finite slab systems and the HEG.

$$V_{\mathrm{xc}}^{\mathrm{LDA}}(x) = \frac{\delta E_{\mathrm{xc}}^{\mathrm{LDA}}[n]}{\delta n(x)} = \varepsilon_{\mathrm{xc}}(n(x)) + n(x)\frac{d\varepsilon_{\mathrm{xc}}}{dn}\bigg|_{n(x)}$$

Parameters

pm [object] Parameters object

n [array_like] 1D array of the electron density, indexed as n[space_index]

separate: bool

- True: Split the HEG exchange-correlation potential into separate exchange and correlation terms

- False: Just return the exchange-correlation potential

returns array_like 1D array of the exchange-correlation potential, indexed as v_xc[space_index]

6.1.9 iDEA.LDA_parameters module

These are the parameters for the LDAs developed in *[Entwistle2018]* from finite slab systems and the HEG.

6.1.10 iDEA.NON module

Calculates the ground-state electron density and energy for a system of N non-interacting electrons through solving the Schroedinger equation. If the system is perturbed, the time-dependent electron density and current density are calculated.

`iDEA.NON.`**`calculate_current_density`**(*pm*, *density*)

Calculates the current density from the time-dependent electron density by solving the continuity equation.

$$\frac{\partial n}{\partial t} + \frac{\partial j}{\partial x} = 0$$

Parameters

pm [object] Parameters object

density [array_like] 2D array of the time-dependent density, indexed as density[time_index,space_index]

returns array_like 2D array of the current density, indexed as current_density[time_index,space_index]

`iDEA.NON.`**`construct_A`**(*pm*, *H*)

Constructs the sparse matrix A, used when solving Ax=b in the Crank-Nicholson propagation.

$$A = I + i\frac{\delta t}{2}H$$

Parameters

pm [object] Parameters object

H [array_like] 2D array containing the band elements of the Hamiltonian matrix, indexed as H[band,space_index]

returns sparse_matrix The sparse matrix A

`iDEA.NON.`**`construct_K`**(*pm*)

Stores the band elements of the kinetic energy matrix in lower form. The kinetic energy matrix is constructed using a three-point, five-point or seven-point stencil. This yields an NxN band matrix (where N is the number of grid points). For example with N=6 and a three-point stencil:

$$K = -\frac{1}{2}\frac{d^2}{dx^2} = -\frac{1}{2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\frac{1}{\delta x^2} = [\frac{1}{\delta x^2}, -\frac{1}{2\delta x^2}]$$

Parameters

pm [object] Parameters object

returns array_like 2D array containing the band elements of the kinetic energy matrix, indexed as K[band,space_index]

`iDEA.NON.main`(*parameters*)

Calculates the ground-state of the system. If the system is perturbed, the time evolution of the perturbed system is then calculated.

Parameters

parameters [object] Parameters object

returns object Results object

6.1.11 iDEA.RE module

Calculates the exact Kohn-Sham potential and exchange-correlation potential for a given electron density using the reverse-engineering algorithm. This works for both a ground-state and time-dependent density.

`iDEA.RE.calculate_current_density`(*pm*, *density_ks*)

Calculates the Kohn-Sham electron current density, at time t+dt, from the time-dependent Kohn-Sham electron density by solving the continuity equation.

$$\frac{\partial n_{\mathrm{KS}}}{\partial t} + \nabla \cdot j_{\mathrm{KS}} = 0$$

Parameters

pm [object] Parameters object

density_ks [array_like] 2D array of the time-dependent Kohn-Sham electron density, at time t and t+dt, indexed as density_ks[time_index, space_index]

returns array_like 1D array of the Kohn-Sham electron current density, at time t+dt, indexed as current_density_ks[space_index]

`iDEA.RE.calculate_ground_state`(*pm*, *approx*, *density_approx*, *v_ext*, *K*)

Calculates the exact ground-state Kohn-Sham potential by solving the ground-state Kohn-Sham equations and iteratively correcting v_ks. The exact ground-state Kohn-Sham eigenfunctions, eigenenergies and electron density are then calculated.

$$V_{\mathrm{KS}}(x) \rightarrow V_{\mathrm{KS}}(x) + \mu[n_{\mathrm{KS}}^{p}(x) - n_{\mathrm{approx}}^{p}(x)]$$

Parameters

pm [object] Parameters object

approx [string] The approximation used to calculate the electron density

density_approx [array_like] 1D array of the ground-state electron density from the approximation, indexed as density_approx[space_index]

v_ext [array_like] 1D array of the unperturbed external potential, indexed as v_ext[space_index]

K [array_like] 2D array of the kinetic energy matrix, index as K[band,space_index]

returns array_like and array_like and array_like and array_like and Boolean 1D array of the ground-state Kohn-Sham potential, indexed as v_ks[space_index]. 1D array of the ground-state Kohn-Sham electron density, indexed as density_ks[space_index].

2D array of the ground-state Kohn-Sham eigenfunctions, indexed as wavefunctions_ks[space_index,eigenfunction]. 1D array containing the ground-state Kohn-Sham eigenenergies, indexed as energies_ks[eigenenergies]. Boolean - True if file containing exact Kohn-Sham potential is found, False if file is not found.

iDEA.RE.**calculate_hartree_energy**(*pm*, *density_ks*, *v_h*)

Calculates the Hartree energy of the ground-state system.

$$E_{\mathrm{H}} = \frac{1}{2}\int\int U(x,x')n(x)n(x')dxdx' = \frac{1}{2}\int V_{\mathrm{H}}(x)n(x)dx$$

Parameters

pm [object] Parameters object

density_ks [array_like] 1D array of the ground-state Kohn-Sham electron density, indexed as density_ks[space_index]

v_h [array_like] 1D array of the ground-state Hartree potential, indexed as v_h[space_index]

returns float The Hartree energy of the ground-state system

iDEA.RE.**calculate_hartree_potential**(*pm*, *density_ks*)

Calculates the Hartree potential for a given electron density.

$$V_{\mathrm{H}}(x) = \int U(x,x')n(x')dx'$$

Parameters

pm [object] Parameters object

density_ks [array_like] 1D array of the Kohn-Sham electron density, either the ground-state or at a particular time step, indexed as density_ks[space_index]

returns array_like 1D array of the Hartree potential for a given electron density, indexed as v_h[space_index]

iDEA.RE.**calculate_time_dependence**(*pm*, *A_initial*, *momentum*, *A_ks*, *damping*, *wavefunctions_ks*, *density_ks*, *density_approx*, *current_density_approx*)

Calculates the exact time-dependent Kohn-Sham vector potential, at time t+dt, by solving the time-dependent Kohn-Sham equations and iteratively correcting A_ks. The exact time-dependent Kohn-Sham eigenfunctions, electron density and electron current density are then calculated.

$$A_{\mathrm{KS}}(x,t) \rightarrow A_{\mathrm{KS}}(x,t) + \nu\left[\frac{j_{\mathrm{KS}}(x,t) - j_{\mathrm{approx}}(x,t)}{n_{\mathrm{approx}}(x,t) + a}\right]$$

Parameters

pm [object] Parameters object

A_initial [sparse_matrix] The sparse matrix A at t=0, A_initial, used when solving the equation Ax=b

momentum [array_like] 1D array of the lower band elements of the momentum matrix, index as momentum[band]

A_ks [array_like] 1D array of the time-dependent Kohn-Sham vector potential, at time t+dt, indexed as A_ks[space_index]

damping [array_like] 1D array of the damping function used to filter out noise, indexed as damping[frequency_index]

wavefunctions_ks [array_like] 2D array of the time-dependent Kohn-Sham eigenfunctions, at time t, indexed as wavefunctions_ks[space_index,eigenfunction]

density_ks [array_like] 2D array of the time-dependent Kohn-Sham electron density, at time t and t+dt, indexed as density_ks[time_index, space_index]

density_approx [array_like] 1D array of the time-dependent electron density from the approximation, at time t+dt, indexed as density_approx[space_index]

current_density_approx [array_like] 1D array of the electron current density from the approximation, at time t+dt, indexed as current_density_approx[space_index]

returns array_like and array_like and array_like and array_like and float

and float 1D array of the time-dependent Kohn-Sham vector potential, at time t+dt, indexed as A_ks[space_index]. 1D array of the time-dependent Kohn-Sham electron density, at time t+dt, indexed as density_ks[space_index]. 1D array of the Kohn-Sham electron current density, at time t+dt, indexed as current_density_ks[space_index]. 2D array of the time-dependent Kohn-Sham eigenfunctions, at time t+dt, indexed as wavefunctions_ks[space_index,eigenfunction]. The error between the Kohn-Sham electron density and electron density from the approximation. The error between the Kohn-Sham electron current density and electron current density from the approximation.

iDEA.RE.**calculate_xc_energy**(*pm*, *approx*, *density_ks*, *v_h*, *v_xc*, *energies_ks*)

Calculates the exchange-correlation energy of the ground-state system.

$$E_{\mathrm{xc}} = E_{\mathrm{total}} - \sum_{j=1}^{N} \varepsilon_j + \int \left[\frac{1}{2}V_{\mathrm{H}}(x) + V_{\mathrm{xc}}(x)\right] n_{\mathrm{KS}}(x)dx$$

Parameters

pm [object] Parameters object

approx [string] The approximation used to calculate the electron density

density_ks [array_like] 1D array of the ground-state Kohn-Sham electron density, indexed as density_ks[space_index]

v_h [array_like] 1D array of the ground-state Hartree potential, indexed as v_h[space_index]

v_xc [array_like] 1D array of the ground-state exchange-correlation potential, indexed as v_xc[space_index]

energies_ks [array_like] 1D array containing the ground-state Kohn-Sham eigenenergies, indexed as energies_ks[eigenenergies]

returns float The exchange-correlation energy of the ground-state system

iDEA.RE.**construct_A**(*pm*, *A_initial*, *A_ks*, *momentum*)

Constructs the sparse matrix A at time t.

$$\begin{aligned} A &= I + i\frac{\delta t}{2}H \\ H &= \frac{1}{2}(p - A_{\mathrm{KS}})^2 + V_{\mathrm{KS}}(t=0) \\ &= K + V_{\mathrm{KS}}(t=0) + \frac{A_{\mathrm{KS}}^2}{2} - \frac{pA_{\mathrm{KS}}}{2} - \frac{A_{\mathrm{KS}}p}{2} \\ &= H(t=0) + \frac{A_{\mathrm{KS}}^2}{2} - \frac{pA_{\mathrm{KS}}}{2} - \frac{A_{\mathrm{KS}}p}{2} \end{aligned}$$

Parameters

pm [object] Parameters object

A_initial [sparse_matrix] The sparse matrix A at t=0

A_ks [array_like] 1D array of the time-dependent Kohn-Sham vector potential, at time t, indexed as A_ks[space_index]

momentum [array_like] 1D array of the lower band elements of the momentum matrix, index as momentum[band,space_index]

returns sparse_matrix The sparse matrix A at time t

`iDEA.RE.`**`construct_A_initial`**(*pm*, *K*, *v_ks*)

Constructs the sparse matrix A at t=0, once the external perturbation has been applied.

$$A_{\text{initial}} = I + i\frac{\delta t}{2}H(t=0)$$
$$H(t=0) = K + V_{\text{KS}}(t=0)$$

Parameters

pm [object] Parameters object

K [array_like] 2D array of the kinetic energy matrix, index as K[band,space_index]

v_ks [array_like] 1D array of the ground-state Kohn-Sham potential + the external perturbation, indexed as v_ks[space_index]

returns sparse_matrix The sparse matrix A at t=0, A_initial, used when solving the equation Ax=b

`iDEA.RE.`**`construct_K`**(*pm*)

Stores the band elements of the kinetic energy matrix in lower form. The kinetic energy matrix is constructed using a three-point, five-point or seven-point stencil. This yields an NxN band matrix (where N is the number of grid points). For example with N=6 and a three-point stencil:

$$K = -\frac{1}{2}\frac{d^2}{dx^2} = -\frac{1}{2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\frac{1}{\delta x^2} = [\frac{1}{\delta x^2}, -\frac{1}{2\delta x^2}]$$

Parameters

pm [object] Parameters object

returns array_like 2D array containing the band elements of the kinetic energy matrix, indexed as K[band,space_index]

`iDEA.RE.`**`construct_damping`**(*pm*)

Stores the damping function which is used to filter out noise in the Kohn-Sham vector potential.

$$f_{\text{damping}}(x) = e^{-10^{-12}(\beta x)^{\sigma}}$$

Parameters

pm [object] Parameters object

returns array_like 1D array of the damping function used to filter out noise, indexed as damping[frequency_index] `iDEA.RE.`**`construct_momentum`**(*pm*)

Stores the band elements of the momentum matrix in lower form. The momentum matrix is constructed using a five-point or seven-point stencil. This yields an NxN band matrix (where N is the number of grid points). For example with N=6 and a five-point stencil:

$$p = -i\frac{d}{dx} = -\frac{i}{12}\begin{pmatrix} 0 & 8 & -1 & 0 & 0 & 0 \\ -8 & 0 & 8 & -1 & 0 & 0 \\ 1 & -8 & 0 & 8 & -1 & 0 \\ 0 & 1 & -8 & 0 & 8 & -1 \\ 0 & 0 & 1 & -8 & 0 & 8 \\ 0 & 0 & 0 & 1 & -8 & 0 \end{pmatrix}\frac{1}{\delta x} = \frac{1}{\delta x}\left[0, \frac{2}{3}, -\frac{1}{12}\right]$$

Parameters

pm [object] Parameters object

returns array_like 1D array of the lower band elements of the momentum matrix, index as momentum[band]

`iDEA.RE.`**`filter_noise`**(*pm*, *A_ks*, *damping*)

Filters out noise in the Kohn-Sham vector potential by suppressing high-frequency terms in the Fourier transform.

Parameters

pm [object] Parameters object

A_ks [array_like] 1D array of the time-dependent Kohn-Sham vector potential, at time t+dt, indexed as A_ks[space_index]

damping [array_like] 1D array of the damping function used to filter out noise, indexed as damping[frequency_index]

returns array_like 1D array of the time-dependent Kohn-Sham vector potential, at time t+dt, indexed as A_ks[space_index]

`iDEA.RE.`**`main`**(*parameters*, *approx*)

Calculates the exact Kohn-Sham potential and exchange-correlation potential for a given electron density using the reverse-engineering algorithm. This works for both a ground-state and time-dependent system.

Parameters

parameters [object] Parameters object

approx [string] The approximation used to calculate the electron density

returns object Results object

`iDEA.RE.`**`read_current_density`**(*pm*, *approx*)

Reads in the electron current density that was calculated using the selected approximation.

Parameters

pm [object] Parameters object

approx [string] The approximation used to calculate the electron current density

returns array_like 2D array of the electron current density from the approximation, indexed as current_density_approx[time_index,space_index]

`iDEA.RE.`**`read_density`**(*pm*, *approx*)

Reads in the electron density that was calculated using the selected approximation.

Parameters

pm [objectmath] Parameters object

approx [string] The approximation used to calculate the electron density

returns array_like 2D array of the ground-state/time-dependent electron density from the approximation, indexed as density_approx[time_index,space_index]

`iDEA.RE.`**`remove_gauge`**(*pm*, *A_ks*, *v_ks*, *v_ks_gs*)

Removes the gauge transformation that was applied to the Kohn-Sham potential, so that it becomes a fully scalar quantity.

$$V_{\mathrm{KS}}(x,t) \rightarrow V_{\mathrm{KS}}(x,t) + \int_{-x_{\max}}^{x} \frac{\partial A_{\mathrm{KS}}(x',t)}{\partial t} dx'$$

Parameters

pm [object] Parameters object

A_ks [array_like] 2D array of the time-dependent Kohn-Sham vector potential, at time t and t+dt, indexed as A_ks[time_index, space_index]

v_ks [array_like] 1D array of the time-dependent Kohn-Sham potential, at time t+dt, indexed as v_ks[space_index]

v_ks_gs [array_like] 1D array of the ground-state Kohn-Sham potential, indexed as v_ks[space_index]

returns array_like 1D array of the time-dependent Kohn-Sham potential, at time t +dt, indexed as v_ks[space_index]

`iDEA.RE.`**`solve_gsks_equations`**(*pm*, *hamiltonian*)

Solves the ground-state Kohn-Sham equations to find the ground-state Kohn-Sham eigenfunctions, energies and electron density.

$$\hat{H}\phi_j(x) = \varepsilon_j \phi_j(x)$$
$$n_{\mathrm{KS}}(x) = \sum_{j=1}^{N} |\phi_j(x)|^2$$

Parameters

pm [object] Parameters object

hamiltonian [array_like] 2D array of the Hamiltonian matrix, index as K[band,space_index]

returns array_like and array_like and array_like 2D array of the ground-state Kohn-Sham eigenfunctions, indexed as wavefunctions_ks[space_index,eigenfunction]. 1D array containing the ground-state Kohn-Sham eigenenergies, indexed as energies_ks[eigenenergies]. 1D array of the ground-state Kohn-Sham electron density, indexed as density_ks[space_index].

`iDEA.RE.`**`solve_tdks_equations`**(*pm*, *A*, *wavefunctions_ks*)

Solves the time-dependent Kohn-Sham equations to find the time-dependent Kohn-Sham eigenfunctions and electron density.

$$\hat{H}\phi_j(x,t) = i\frac{\partial \phi_j(x,t)}{\partial t}$$
$$n_{\mathrm{KS}}(x,t) = \sum_{j=1}^{N} |\phi_j(x,t)|^2$$

Parameters

pm [object] Parameters object

A [sparse_matrix] The sparse matrix A, used when solving the equation Ax=b

wavefunctions_ks [array_like] 2D array of the time-dependent Kohn-Sham eigenfunctions, at time t+dt, indexed as wavefunctions_ks[space_index,eigenfunction]

returns array_like and array_like 1D array of the time-dependent Kohn-Sham electron density, at time t+dt, indexed as density_ks[space_index]. 2D array of the time-dependent Kohn-Sham eigenfunctions, at time t+dt, indexed as wavefunctions_ks[space_index,eigenfunction]

`iDEA.RE.`**`xc_correction`**`(`*pm*, *v_xc*`)`

Calculates an approximation to the constant that needs to be added to the exchange-correlation potential so that it asymptotically approaches zero at large $|x|$. The approximate error (standard deviation) on the constant is also calculated.

$$V_{\mathrm{xc}}(x) \rightarrow V_{\mathrm{xc}}(x) + a \ , \ \ \text{s.t.} \lim_{|x| \rightarrow \infty} V_{\mathrm{xc}}(x) = 0$$

Parameters

pm [object] Parameters object

v_xc [array_like] 1D array of the ground-state exchange-correlation potential, indexed as v_xc[space_index]

returns float and float An approximation to the constant that needs to be added to the exchange-correlation potential. The approximate error (standard deviation) on the constant.

`iDEA.RE.`**`xc_fit`**`(`*grid*, *correction*`)`

Applies a fit to the exchange-correlation potential over a specified range near the edge of the system's grid to determine the correction that needs to be applied to give the correct asymptotic behaviour at large $|x|$

$$V_{\mathrm{xc}}(x) \approx \frac{1}{x} + a$$

Parameters

grid [array_like] 1D array of the spatial grid over a specified range near the edge of the system

correction [float] An approximation to the constant that needs to be added to the exchange-correlation potential

returns array_like A fit to the exchange-correlation potential over the specified range

6.1.12 iDEA.RE_cython module

Contains the cython modules that are called within RE. Cython is used for operations that are very expensive to do in Python, and performance speeds are close to C.

`iDEA.RE_cython.`**`continuity_eqn`**`()`

Calculates the electron current density of the system for a particular time step by solving the continuity equation.

Parameters

pm [object] Parameters object

density_new [array_like] 1D array of the electron density at time t

density_old [array_like] 1D array of the electron density at time t-dt

returns array_like 1D array of the electron current density at time t

6.1.13 iDEA.info module

Contains information on version, authors, etc.

`iDEA.info.`**`get_sha1`**`()`

Returns sha1 hash of last commit from git

Works only, if the code resides inside a git repository. Returns None otherwise.

6.1.14 iDEA.input module

Stores input parameters for iDEA calculations.

`class` `iDEA.input.`**`Input`**

Bases: `object`

Stores variables of input parameters file

Includes automatic generation of dependent variables, checking of input parameters, printing back to file and more.

`__dict__ = mappingproxy({'__module__': 'iDEA.input', '__doc__': 'Stores variables of`

`__init__``()`

Sets default values of some properties.

`__module__ = 'iDEA.input'`

`__str__``()`

Prints different sections in input file

`__weakref__`

list of weak references to the object (if defined)

`check``()`

Checks validity of input parameters.

`execute``()`

Run this job

`classmethod from_python_file``(`*filename*`)`

Create Input from Python script.

`make_dirs``()`

Set up ouput directory structure

`output_dir`

Returns full path to output directory

`priority_dict = {'default': 1, 'high': 0, 'low': 2}`

`read_from_python_file``(`*filename*`)`

Update Input from Python script.

`setup_space``()`

Prepares for performing calculations

precomputes quantities on grids, etc.

`sprint``(`*string=”*, *priority=1*, *newline=True*, *refresh=5e-06*, *savelog=True*`)`

Customized print function

Prints to screen and appends to log.

If newline == False, overwrites last line, but refreshes only every refresh seconds.

Parameters

string [string] string to be printed

priority: int priority of message, possible values are 0: debug 1: normal 2: important

newline [bool] If False, overwrite the last line

refresh [float] If newline == False, print only every "refresh" seconds

savelog [bool] If True, save string to log file

class iDEA.input.**InputSection**

Bases: `object`

Generic section of input file

`__dict__ = mappingproxy({'__module__': 'iDEA.input', '__doc__': 'Generic section of`

`__module__ = 'iDEA.input'`

`__str__`()

Print variables of section and their values

`__weakref__`

list of weak references to the object (if defined)

class iDEA.input.**SpaceGrid**(*pm*)

Bases: `object`

Stores basic real space arrays

These arrays should be helpful in many types of iDEA calculations. Storing them in the Input object avoids having to recompute them and reduces code duplication.

`__dict__ = mappingproxy({'__module__': 'iDEA.input', '__doc__': 'Stores basic real s`

`__init__`(*pm*)

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'iDEA.input'`

`__str__`()

Print variables of section and their values

`__weakref__`

list of weak references to the object (if defined)

class iDEA.input.**SystemSection**

Bases: *iDEA.input.InputSection*

System section of input file

Includes some derived quantities.

`__module__ = 'iDEA.input'`

`deltat`

Spacing of temporal grid

`deltax`

Spacing of real space grid

`grid_points`

Real space grid

`iDEA.input.`**`input_string`**(*key*, *value*)
Prints a line of the input file

6.1.15 iDEA.minimize module

Direct minimisation of the Hamiltonian

`class` `iDEA.minimize.`**`CGMinimizer`**(*pm*, *total_energy=None*, *nstates=None*, *cg_restart=3*, *line_fit='quadratic'*)

Bases: `object`

Performs conjugate gradient minimization

Performs Pulay mixing with Kerker preconditioner, as described on pp 1071 of *[Payne1992]*

`__dict__ = mappingproxy({'__module__': 'iDEA.minimize', '__doc__': 'Performs conjuga`

`__init__`(*pm*, *total_energy=None*, *nstates=None*, *cg_restart=3*, *line_fit='quadratic'*)
Initializes variables

Parameters

pm: object input parameters

total_energy: callable function f(pm, waves) that returns total energy

nstates: int how many states to retain. currently, this must equal the number of occupied states (for unoccupied states need to re-diagonalize)

cg_restart: int cg history is restarted every cg_restart steps

line_fit: str select method for line-search 'quadratic' (default), 'quadratic-der' or 'trigonometric'

`__module__ = 'iDEA.minimize'`

`__weakref__`
list of weak references to the object (if defined)

`braket`(*bra=None*, *O=None*, *ket=None*)
Compute braket with operator O

bra and ket may hold multiple vectors or may be empty. Variants:

Parameters

bra: array_like (grid, nwf) lhs of braket

O: array_like (grid, grid) operator. defaults to identity matrix

ket: array_like (grid, nwf) rhs of braket

`conjugate_directions`(*steepest_dirs*, *wfs*)
Compute conjugate gradient descent for one state

Updates internal arrays accordingly

See eqns (5.8-9) in *[Payne1992]*

Parameters

steepest_dirs: array_like steepest-descent directions (grid, nwf)

wfs: array_like wave functions (grid, nwf)

Returns

cg_dirs: array_like conjugate directions (grid, nwf)

exact_dirs(*wfs*, *H*)
Search direction from exact diagonalization

Just for testing purposes (you can easily end up with multiple minima along the exact search directions)

line_search(*wfs*, *dirs*, *H*, *mode*)
Performs a line search along cg direction

Trying to minimize total energy

$$E(s) = \langle\psi(s)|H[\psi(s)]|\psi(s)\rangle$$

steepest_dirs(*wfs*, *H*)
Compute steepest descent directions

Compute steepest descent directions and project out components pointing along other orbitals (equivalent to steepest descent with the proper Lagrange multipliers).

See eqns (5.10), (5.12) in *[Payne1992]*

Parameters

H: array_like Hamiltonian matrix (grid,grid)

wavefunctions: array_like wave function array (grid, nwf)

Returns

steepest_orth: array_like steepest descent directions (grid, nwf)

step(*wfs*, *H*)
Performs one cg step

After each step, the Hamiltonian should be recomputed using the updated wave functions.

Note that we currently don't enforce the wave functions to remain eigenfunctions of the Hamiltonian. This should not matter for the total energy but means we need to performa a diagonalisation at the very end.

Parameters

wfs: array_like (grid, nwf) input wave functions

H: array_like input Hamiltonian

Returns

wfs: array_like (grid, nwf) updated wave functions

subspace_diagonalization(*v*, *H*)
Diagonalise suspace of wfs

Parameters

v: array_like (grid, nwf) array of orthonormal vectors

H: array_like (grid,grid) Hamiltonian matrix

Returns

v_rot: array_like (grid, nwf) array of orthonormal eigenvectors of H (or at least close to eigenvectors)

total_energy(*wfs*)
Compute total energy for given wave function

This method must be provided by the calling module and is initialized in the constructor.

class iDEA.minimize.**DiagMinimizer**(*pm*, *total_energy=None*)

Bases: `object`

Performs minimization using exact diagonalisation

This would be too slow for ab initio codes but is something we can afford in the iDEA world.

Not yet fully implemented though (still missing analytic derivatives and a proper line search algorithm).

`__dict__ = mappingproxy({'__module__': 'iDEA.minimize', '__doc__': 'Performs minimiz`

`__init__`(*pm*, *total_energy=None*)

Initializes variables

Parameters

pm: object input parameters

total_energy: callable function f(pm, waves) that returns total energy

nstates: int how many states to retain. currently, this must equal the number of occupied states (for unoccupied states need to re-diagonalize)

`__module__ = 'iDEA.minimize'`

`__weakref__`

list of weak references to the object (if defined)

`h_step`(*H0*, *H1*)

Performs one minimisation step

Parameters

H0: array_like input Hamiltonian to be mixed (banded form)

H1: array_like output Hamiltonian to be mixed (banded form)

Returns

H: array_like mixed hamiltonian (banded form)

`total_energy`(*wfs*)

Compute total energy for given wave function

This method must be provided by the calling module and is initialized in the constructor.

iDEA.minimize.**orthonormalize**(*v*)

Return orthonormalized set of vectors

Return orthonormal set of vectors that spans the same space as the input vectors.

Parameters

v: array_like (n, m) array of m vectors in n-dimensional space

6.1.16 iDEA.mix module

Mixing schemes for self-consistent calculations

class iDEA.mix.**PulayMixer**(*pm*, *order*, *preconditioner=None*)

Bases: `object`

Performs Pulay mixing

Can perform Pulay mixing with Kerker preconditioning as described on p.34 of *[Kresse1996]* Can also be combined with other preconditioners (see precondition.py).

`__dict__ = mappingproxy({'__module__': 'iDEA.mix', '__doc__': 'Performs Pulay mixing`

`__init__`(*pm*, *order*, *preconditioner=None*)
Initializes variables

Parameters

order: int order of Pulay mixing (how many densities to keep in memory)

pm: object input parameters

preconditioner: string May be None, 'kerker' or 'rpa'

`__module__ = 'iDEA.mix'`

`__weakref__`
list of weak references to the object (if defined)

`compute_coefficients`(*m*, *ncoef*)
Computes mixing coefficients

See *[Kresse1996]* equations (87) - (90)

$$A_{ij} = \langle R[\rho_{in}^j] | R[\rho_{in}^i \rangle$$
$$\bar{A}_{ij} = \langle \Delta R^j | \Delta R^i \rangle$$

See *[Kresse1996]* equation (92)

Parameters

m: int array index for non-delta quantities

ncoef: int number of coefficients to compute

`mix`(*den_in*, *den_out*, *eigv=None*, *eigf=None*)
Compute mix of densities

Computes new input density rho_in^{m+1}, where the index m corresponds to the index m used in *[Kresse1996]* on pp 33-34.

Parameters

den_in: array_like input density

den_out: array_like output density

`precondition`(*f*, *eigv*, *eigf*)
Return preconditioned f

`update_arrays`(*m*, *den_in*, *den_out*)
Updates densities and residuals

We need to store:

- delta-quantities from i=1 up to m-1
- den_in i=m-1, m
- r i=m-1, m

In order to get Pulay started, we do one Kerker-only step (step 0).

Note: When self.step becomes larger than self.order, we overwrite data that is no longer needed.

Parameters

m: int array index for non-delta quantities

den_in: array_like input density

den_out: array_like output density

6.1.17 iDEA.plot module

Plotting output quantities of iDEA

`iDEA.plot.`**`read_quantity`**(*pm*, *name*)

Read a file from a pickle file in (/raw)

Parameters

pm: parameters object parameters object

name: string name of pickle file in (/raw) (e.g 'gs_ext_den')

returns array_like data extracted from pickle file

`iDEA.plot.`**`to_anim`**(*pm*, *names*, *data*, *td*, *dim*, *file_name=None*, *step=1*)

Outputs data to a .mp4 file in (/animations)

Parameters

pm: parameters object parameters object

names: list of string names of the data to be saved (e.g 'gs_ext_den')

data: list of array_like list of arrays to be plotted

td: bool True: Time-dependent data, False: ground-state data

dim: int number of dimentions of the data (0,1,2 or 3) (eg gs_ext_E would be 0, gs_ext_den would be 1)

file_name: string name of output file (if None will be saved as default name e.g 'gs_ext_den.dat')

step: int number of frames to skip when animating

`iDEA.plot.`**`to_data`**(*pm*, *names*, *data*, *td*, *dim*, *file_name=None*, *timestep=0*)

Outputs data to a .dat file in (/data)

Parameters

pm: parameters object parameters object

names: list of string names of the data to be saved (e.g 'gs_ext_den')

data: list of array_like list of arrays to be plotted

td: bool True: Time-dependent data, False: ground-state data

dim: int number of dimentions of the data (0,1,2 or 3) (eg gs_ext_E would be 0, gs_ext_den would be 1)

file_name: string name of output file (if None will be saved as default name e.g 'gs_ext_den.dat')

timestep: int if td=True or data=3D specify the timestep to be saved

`iDEA.plot.`**`to_plot`**(*pm*, *names*, *data*, *td*, *dim*, *file_name=None*, *timestep=0*)

Outputs data to a .pdf file in (/plots)

Parameters

pm: parameters object parameters object

names: list of string names of the data to be saved (e.g 'gs_ext_den')

data: list of array_like list of arrays to be plotted

td: bool True: Time-dependent data, False: ground-state data

dim: int number of dimentions of the data (0,1,2 or 3) (eg gs_ext_E would be 0, gs_ext_den would be 1)

file_name: string name of output file (if None will be saved as default name e.g 'gs_ext_den.pdf')

timestep: int if td=True or data=3D specify the timestep to be saved

6.1.18 iDEA.precondition module

Preconditioners for self-consistent calculations.

Can be used in conjunction with mixing schemes (see mix.py).

class `iDEA.precondition.`**`KerkerPreconditioner`**(*pm*)

Bases: `object`

Performs Kerker preconditioning

Performs Kerker preconditioning, as described on p.34 of *[Kresse1996]*

`__dict__ = mappingproxy({'__module__': 'iDEA.precondition', '__doc__': 'Performs Ker`

`__init__`(*pm*)

Initializes variables

Parameters

pm: object input parameters

`__module__ = 'iDEA.precondition'`

`__weakref__`

list of weak references to the object (if defined)

`precondition`(*f*, *eigv*, *eigf*)

Return preconditioned f

class `iDEA.precondition.`**`RPAPreconditioner`**(*pm*)

Bases: `object`

Performs preconditioning using RPA dielectric function

The static dielectric function as a function of x and x' is computed in the Hartree approximation.

`__dict__ = mappingproxy({'__module__': 'iDEA.precondition', '__doc__': "Performs pre`

`__init__`(*pm*)

Initializes variables

Parameters

pm: object input parameters

`__module__ = 'iDEA.precondition'`

`__weakref__`

list of weak references to the object (if defined)

chi (*eigv*, *eigf*)
Computes RPA polarizability

The static, non-local polarisability (aka density-potential response) in the Hartree approximation (often called RPA):

$$\chi^0(x,x') = \sum_j^{\prime} \sum_k^{\prime\prime} \phi_j(x)\phi_k^*(x)\phi_j^*(x')\phi_k(x')\frac{2}{\varepsilon_j - \varepsilon_k}$$

where $\sum^{\prime}$ sums over occupied states and $\sum^{\prime\prime}$ sums over empty states

See also https://wiki.fysik.dtu.dk/gpaw/documentation/tddft/dielectric_response.html

Parameters

eigv: array_like array of eigenvalues

eigf: array_like array of eigenfunctions

Returns

epsilon: array_like dielectric matrix in real space

precondition (*r*, *eigv*, *eigf*)
Preconditioning using RPA dielectric matrix

$$\frac{\delta V(x)}{\delta \rho(x')} = DXC(x)\delta(x-x') + v(x-x')$$

Parameters

r: array_like array of residuals to be preconditioned

eigv: array_like array of eigenvalues

eigf: array_like array of eigenfunctions

class iDEA.precondition.**StubPreconditioner** (*pm*)
Bases: `object`

Performs no preconditioning

__dict__ = mappingproxy({'__module__': 'iDEA.precondition', '__doc__': 'Performs no

__init__ (*pm*)
Initializes variables

Parameters

pm: object input parameters

__module__ = 'iDEA.precondition'

__weakref__
list of weak references to the object (if defined)

precondition (*f*, *eigv*, *eigf*)
Return preconditioned f

6.1.19 iDEA.results module

Bundles and saves iDEA results

class iDEA.results.**Results**

Bases: `object`

Container for results.

A convenient container for storing, reading and saving the results of a calculation.

Usage:

```
res = Results()
res.add(my_result, 'my_name')
res.my_name  # now contains my_result
res.save(pm)  # saves to disk + keeps track

res.add(my_result2, 'my_name2')
res.save(pm)  # saves only my_result2 to disk
```

`__dict__ = mappingproxy({'__module__': 'iDEA.results', '__doc__': "Container for res`

`__init__`()

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'iDEA.results'`

`__weakref__`

list of weak references to the object (if defined)

`_not_saved`

Returns list of results not yet saved to disk.

`add`(*results*, *name*)

Add results to the container.

Note: Existing results are overwritten.

`add_pickled_data`(*name*, *pm*, *dir=None*)

Read results from pickle file and adds to results.

Parameters

name [string] name of results to be read (filepath = raw/name.db)

pm [object] iDEA.input.Input object

dir [string] directory where result is stored default: pm.output_dir + '/raw'

`calc_dict = {'gs': 'ground state', 'td': 'time-dependent'}`

`static label`(*shortname*)

returns full label for shortname of result.

Expand shortname used for quantities saved by iDEA. E.g. 'gs_non_den' => 'ground state rho (non-interacting)'

`method_dict = {'ext': 'exact', 'hf': 'Hartree-Fock', 'lda': 'LDA', 'non': 'non-int`

`quantity_dict = {'S': '$\\Sigma$', 'Sc': '$\\Sigma_{c}$', 'Sx': '$\\Sigma_{x}$', 'Sx`

`static read`(*name*, *pm*, *dir=None*)

Reads and returns results from pickle file

Parameters

name [string] name of result to be read (filepath = raw/name.db)

pm [object] iDEA.input.Input object

dir [string] directory where result is stored default: pm.output_dir + '/raw'

Returns data

save (*pm*, *dir=None*, *list=None*)
Save results to disk.

Note: Saves only results that haven't been saved before.

Parameters

pm [object] iDEA.input.Input object

dir [string] directory where to save results default: pm.output_dir + '/raw'

verbosity [string] additional info will be printed for verbosity 'high'

list [array_like] if given, saves listed results if not set, saves results that haven't been saved before

save_hdf5 (*pm*, *dir=None*, *list=None*, *f=None*)
Save results to HDF5 database.

This requires the h5py python package.

Parameters

pm [object] iDEA.input.Input object

dir [string] directory where to save results default: pm.output_dir + '/raw'

verbosity [string] additional info will be printed for verbosity 'high'

list [array_like] if set, only the listed results will be saved

f [HDF5 handle] handle of HDF5 file (or group) to write to

6.1.20 iDEA.splash module

Prints iDEA logo as splash

`iDEA.splash.`**`draw`**(*pm*)

6.1.21 iDEA.test_EXT1 module

Tests for 1-electron exact calculations in iDEA

class `iDEA.test_EXT1.`**`TestAtom`**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for an atomic-like potential

External potential is a softened atomic-like potential. Testing ground-state case. Testing 3-, 5- and 7-point stencil for the second-derivative.

`__module__ = 'iDEA.test_EXT1'`

setUp()
Sets up atomic system

test_stencil_five()
Test 5-point stencil

test_stencil_seven()
Test 7-point stencil

test_stencil_three()
Test 3-point stencil

class iDEA.test_EXT1.**TestHarmonicOscillator**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for the harmonic oscillator potential

External potential is the harmonic oscillator (this is the default in iDEA). Testing ground-state and time-dependence case.

__module__ = 'iDEA.test_EXT1'

setUp()
Sets up harmonic oscillator system

test_system()
Test ground-state and then real time propagation

6.1.22 iDEA.test_EXT2 module

Tests for 2-electron exact calculations in iDEA

class iDEA.test_EXT2.**TestDoubleWell**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for an asymmetric double-well potential

External potential is an asymmetric double-well potential (System 1 in Hodgson et al. 2016). Testing ground-state interacting case. Testing 3-, 5- and 7-point stencil for the second-derivative. Testing initial wavefunction by starting from the non-interacting orbitals.

__module__ = 'iDEA.test_EXT2'

setUp()
Sets up double-well system

test_stencil_five()
Test 5-point stencil

test_stencil_seven()
Test 7-point stencil

test_stencil_three()
Test 3-point stencil

class iDEA.test_EXT2.**TestHarmonicOscillator**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for the harmonic oscillator potential

External potential is the harmonic oscillator (this is the default in iDEA). Testing both ground-state non-interacting and ground-state interacting case. Testing time-dependent interacting case.

__module__ = 'iDEA.test_EXT2'

setUp()
Sets up harmonic oscillator system

test_interacting_system_1()
Test interacting system

test_non_interacting_system_1()
Test non-interacting system

test_time_dependence()
Test real time propagation

6.1.23 iDEA.test_EXT3 module

Tests for 3-electron exact calculations in iDEA

class iDEA.test_EXT3.**TestDoubleWell**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for an asymmetric double-well potential

External potential is an asymmetric double-well potential (System 1 in Hodgson et al. 2016). Testing ground-state interacting case. Testing 3-, 5- and 7-point stencil for the second-derivative. Testing initial wavefunction by starting from the non-interacting orbitals.

__module__ = 'iDEA.test_EXT3'

setUp()
Sets up double-well system

test_stencil_five()
Test 5-point stencil

test_stencil_seven()
Test 7-point stencil

test_stencil_three()
Test 3-point stencil

class iDEA.test_EXT3.**TestHarmonicOscillator**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Tests for the harmonic oscillator potential

External potential is the harmonic oscillator (this is the default in iDEA). Testing both ground-state non-interacting and ground-state interacting case. Testing time-dependent interacting case.

__module__ = 'iDEA.test_EXT3'

setUp()
Sets up harmonic oscillator system

test_interacting_system_1()
Test interacting system

test_non_interacting_system_1()
Test non-interacting system

test_time_dependence()
Test real time propagation

6.1.24 iDEA.test_HF module

Tests for the local density approximation

class iDEA.test_HF.**HFTestHarmonic**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Tests on the harmonic oscillator potential

__module__ = 'iDEA.test_HF'

setUp()
Sets up harmonic oscillator system

test_1_electron()
Ensures HF is exact for one electron

6.1.25 iDEA.test_HYB module

Tests for the local density approximation

class iDEA.test_HYB.**HYBTestHarmonic**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Tests on the harmonic oscillator potential

__module__ = 'iDEA.test_HYB'

setUp()
Sets up harmonic oscillator system

test_alpha()
Ensures HYB is idential to HF if a=1.0

6.1.26 iDEA.test_LDA module

Tests for the local density approximation

class iDEA.test_LDA.**LDATestHarmonic**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Tests on the harmonic oscillator potential

__module__ = 'iDEA.test_LDA'

setUp()
Sets up harmonic oscillator system

test_banded_hamiltonian_1()
Test construction of Hamiltonian

Hamiltonian is constructed in banded form for speed. This checks that the construction actually works.

test_kinetic_energy_1()
Checks kinetic energy

Constructs Hamiltonian with KS-potential set to zero and computes expectation values.

test_total_energy_1()
Compares total energy computed via two methods

One method uses the energy eigenvalues + correction terms. The other uses it only for the kinetic part, while the rest is computed usin energy functinals.

6.1.27 iDEA.test_NON module

Tests for non-interacting calculations in iDEA

class `iDEA.test_NON.`**`TestAtom`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for an atomic-like potential

External potential is a softened atomic-like potential. Testing ground-state case. Testing 3-, 5- and 7-point stencil for the second-derivative.

`__module__ = 'iDEA.test_NON'`

`setUp`()

Sets up atomic system

`test_stencil_five`()

Test 5-point stencil

`test_stencil_seven`()

Test 7-point stencil

`test_stencil_three`()

Test 3-point stencil

class `iDEA.test_NON.`**`TestHarmonicOscillator`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for the harmonic oscillator potential

External potential is the harmonic oscillator (this is the default in iDEA). Testing ground-state and time-dependence case.

`__module__ = 'iDEA.test_NON'`

`setUp`()

Sets up harmonic oscillator system

`test_system`()

Test ground-state and then real time propagation

6.1.28 iDEA.test_minimize module

Tests for direct minimizers

class `iDEA.test_minimize.`**`TestCG`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for the conjugate gradient minimizer

`__module__ = 'iDEA.test_minimize'`

`setUp`()

Sets up harmonic oscillator system

`test_conjugate`()

Check that gradient is computed correctly

Use conjugate-gradient method on a quadratic function in n-dimensional space, where it is guaranteed to converge in n steps.

Note: This exact condition actually doesn't apply here because we are orthonormalising the vectors (i.e. rotating) after each step. This renders this test a bit pointless… I am currently taking 2*n steps and am still far from machine precision.

Task: minimize the function

$$E(v) = \sum_{i=1}^{2} v_i^* H v_i \triangle_{v_i} E = H v_i$$

for a fixed matrix H and a set of orthonormal vectors v_i.

Note: The dimension n of the vector space is the grid spacing times number of electrons.

test_orthonormalisation()

Testing orthonormalisation of a set of vectors

minimize. This should do the same as Gram-Schmidt

test_steepest_dirs()

Testing orthogonalisation in steepest descent

Just checking that the efficient numpy routines do the same as more straightforward loop-based techniques

class iDEA.test_minimize.**TestCGLDA**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for the conjugate gradient minimizer

On an actual LDA system

__module__ = 'iDEA.test_minimize'

setUp()

Sets up harmonic oscillator system

test_energy_derivative()

Compare analytical total energy derivative vs finite differences

$$\frac{dE}{d\lambda}(\lambda) \approx \frac{E(\lambda) + E(\lambda + \delta)}{\delta}$$

6.1.29 iDEA.test_mix module

Tests for mixing schemes

class iDEA.test_mix.**TestKerker**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for the Kerker preconditioner

__module__ = 'iDEA.test_mix'

setUp()

Sets up harmonic oscillator system

test_screening_length_1()

Testing screening length in Kerker

Check that for infinite screening length, simple mixing is recovered. [Kresse1996] p.34 …

class iDEA.test_mix.**TestPulay**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Tests for the Pulay mixer

`__module__ = 'iDEA.test_mix'`

`setUp``()`
Sets up harmonic oscillator system

`test_array_update_1``()`
Testing internal variables of Pulay mixer

Just checking that the maths works as expected from *[Kresse1996]* p.34 …

`class` `iDEA.test_mix.`**`TestRPA`**`(`*methodName='runTest'*`)`
Bases: `unittest.case.TestCase`

Tests for the RPA preconditioner

`__module__ = 'iDEA.test_mix'`

`setUp``()`
Sets up harmonic oscillator system

`test_chi_1``()`
Testing potential-density response

Testing some basic symmetry properties of the potential-density response and the preconditioning matrices required for density/potential mixing.

6.1.30 iDEA.test_result module

Tests for the result class

`class` `iDEA.test_result.`**`resultsTest`**`(`*methodName='runTest'*`)`
Bases: `unittest.case.TestCase`

Tests results object

`__module__ = 'iDEA.test_result'`

`setUp``()`
Sets up harmonic oscillator system

`test_save_1``()`
Checks that saving works as expected

6.1.31 Module contents

interacting Dynamic Electrons Approach (iDEA)

The iDEA code allows to propagate the time-dependent Schroedinger equation for 2-3 electrons in one-dimensional real space. Compared to other models, such as the Anderson impurity model, this allows us to treat exchange and correlation throughout the system and provides additional flexibility in bridging the gap between model systems and ab initio descriptions.

The iDEA code (interacting Dynamic Electrons Approach) is a Python-Cython software suite developed in Rex Godby's group at the University of York since 2010. It has a central role in a number of research projects related to many-particle quantum mechanics for electrons in matter.

iDEA's main features are:

- Exact solution of the many-particle time-independent Schrödinger equation, including exact exchange and correlation

- Exact solution of the many-particle time-dependent Schrödinger equation, including exact exchange and correlation
- Simplicity achieved using spinless electrons in one dimension
- An arbitrary external potential that may be time-dependent
- Optimisation methods to determine the exact DFT/TDDFT Kohn-Sham potential and energy components
- Implementation of various approximate functionals (established and novel) for comparison

iDEA Contributors: Sean Adamson, Jacob Chapman, Thomas Durrant, Razak Elmaslmane, Mike Entwistle, Rex Godby, Matt Hodgson, Piers Lillystone, Aaron Long, Robbie Oliver, James Ramsden, Ewan Richardson, Matthew Smith, Leopold Talirz and Jack Wetherell

iDEA is released under the MIT license

Bibliography

[Hohenberg1964] "Inhomogeneous Electron Gas" P. Hohenberg and W. Kohn (1964) Phys. Rev. 136, B864

[Kohn1965] "Self-Consistent Equations Including Exchange and Correlation Effects" W. Kohn and L. J. Sham (1965) Phys. Rev. 140, A1133

[Entwistle2018] 13. (t) Entwistle, M. Casula, and R. W. Godby, Comparison of local density functionals based on electron gas and finite systems, Phys. Rev. B 97, 235143 (2018).

[Kresse1996] Kresse, G. & Furthmüller, J. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Computational Materials Science 6, 15–50 (1996). doi: 10.1016/0927-0256(96)00008-0

[Payne1992] Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos, Rev. Mod. Phys. 64, 1045 (1992).

Python Module Index

Index