Jussi Enkovaara
CSC – IT Center for Science, Finland
Outline

- Introduction to GPAW
  - Projector-augmented wave method
  - Basis sets in GPAW
- Using Atomic Simulation Environment and GPAW
- Hands-on exercises

- Lunch

- Time-dependent DFT with GPAW
- Parallel calculations ( + other features by request)
- Hands-on exercises
GPAW

Implementation of projector augmented wave method on
- uniform real-space grids, atomic orbital basis, plane waves
Density-functional theory, time-dependent DFT, many-body perturbation theory, ...
Massively parallelized

wiki.fysik.dtu.dk/gpaw

gpaw-users@listserv.fysik.dtu.dk

GPAW

- Open source software licensed under GPL
  - 20-30 developers in Europe and USA
- The main GPAW references cited ~350 times
- ~300 subscribers in users mailing list
GPAW features

- Total energies, forces, structural optimization
  - analysis of electronic structure
- Excited states, optical spectra
  - Non-adiabatic electron-ion dynamics
- Wide range of XC-potentials (thanks to libxc!)
  - LDAs, GGAs, meta-GGAs, hybrids, DFT+U, vdW, RPA
- Electron transport
- GW-approximation, Bethe-Salpeter equation
- ...
Projector-augmented wave method

- Exact all-electron formalism
- “Pseudopotentials done right”
- Good description over the whole periodic table
- Access to full wave functions and density
- Norm-conserving and ultrasoft pseudopotentials can be derived as approximations to PAW
PAW transformation

Linear transformation from pseudo wave functions to all-electron wave functions

\[ |\psi_k\rangle = \hat{T} |\tilde{\psi}_k\rangle \]

Transformation operator

\[ \hat{T} = 1 + \sum_{a,i} \left( |\phi_i^a\rangle - |\tilde{\phi}_i^a\rangle \right) \langle \tilde{p}_i^a | \]

- \( |\phi_i^a\rangle \) all-electron atomic orbital
- \( |\tilde{\phi}_i^a\rangle \) pseudo atomic orbital
- \( |\tilde{p}_i^a\rangle \) projector function

Core electrons are frozen
Properties of PAW transformation

- Projector functions are localized inside the augmentation spheres
  \[ p_i^a(r) = 0, r > R_a \]

- All-electron orbitals and pseudo orbitals are equal outside the augmentation spheres
  \[ \phi_i^a(r) = \tilde{\phi}_i^a(r), r > R_a \]

- Projector functions are orthogonal to pseudo orbitals
  \[ \langle \tilde{p}_i^a | \tilde{\phi}_j^a \rangle = \delta_{ij} \]

Example: Platinum
PAW transformation

The projector functions and partial waves are constructed from all-electron calculation for spherical symmetric atom.

To be exact, infinite number of projectors and partial waves is needed.

- In practice, 1-2 functions per angular momentum is enough.
One center expansion

Inside augmentation spheres one can define one-center expansions of AE and PS state

$$\psi_n^a(r) = \sum_i P_{in}^a \phi_i^a(r)$$

$$\tilde{\psi}_n^a(r) = \sum_i P_{in}^a \tilde{\phi}_i^a(r)$$

with expansion coefficients

$$P_{in}^a = \langle \tilde{\phi}_i^a | \tilde{\psi}_n \rangle$$

All electron wave function can now be written as

$$\psi_n = \tilde{\psi}_n + \sum_a (\psi_n^a - \tilde{\psi}_n^a)$$
PAW expectation values

Within frozen core approximation expectation value of operator $\hat{O}$ is

$$\langle \hat{O} \rangle = \sum_{n}^{val} f_{n} \langle \psi_{n} | \hat{O} | \psi_{n} \rangle + \sum_{a} \sum_{\alpha} \langle \phi_{\alpha}^{a, \text{core}} | \hat{O} | \phi_{\alpha}^{a, \text{core}} \rangle$$

By inserting the PAW expression, one obtains for (semi)local operators

$$\langle \psi_{n} | \hat{O} | \psi_{n} \rangle = \langle \tilde{\psi}_{n} | \hat{O} | \tilde{\psi}_{n} \rangle + \sum_{a} \left( \langle \psi_{n}^{a} | \hat{O} | \psi_{n}^{a} \rangle - \langle \tilde{\psi}_{n}^{a} | \hat{O} | \tilde{\psi}_{n}^{a} \rangle \right)$$

$$= \hat{O} + \sum_{a} \left( O^{a} - \hat{O}^{a} \right)$$
PAW Hamiltonian

PAW Hamiltonian can be written as

\[ \hat{H} = -\frac{1}{2} \nabla^2 + \tilde{v}(\mathbf{r}) + \sum_a \sum_{i_1 i_2} \langle \tilde{\psi}_{i_1}^a | \Delta H_{i_1 i_2}^a | \tilde{\psi}_{i_2}^a \rangle \]

Pseudo wave functions are orthonormal only with respect to overlap operator

\[ \langle \psi_i | \psi_j \rangle = \langle \tilde{\psi}_i | \hat{T}^\dagger \hat{T} | \tilde{\psi}_j \rangle = \langle \tilde{\psi}_i | \hat{S} | \tilde{\psi}_j \rangle = \delta_{ij} \]

Generalized eigenvalue equation

\[ \hat{H} \tilde{\psi}_n = \epsilon_n \hat{S} \tilde{\psi}_n \]
Approximations in PAW

- Finite number of projectors
  - typically two projectors per angular momentum are used
- Truncated angular momentum expansions
- Overlapping augmentation spheres
- Frozen core
PAW setup

A set of \( \phi^a, \tilde{\phi}^a, \tilde{p}^a, n_c^a, \tilde{n}_c^a, v^a \) for a single atom constitutes a PAW setup.

Setups are generated for individual atoms.

The actual PAW calculations use pregenerated setups.

Setup testing is non-trivial and time-consuming:
- correct properties for the particular atom
- transferability in different molecules and solids
Basis sets in GPAW

- Real-space grids
- Localized atomic orbital basis
- Plane waves
Real-space grids

Wave functions, electron densities, and potentials are represented on grids.

Single parameter, grid spacing $h$

Accuracy of calculation can be improved systematically

Derivatives by finite differences
Boundary conditions

- Real-space description allows flexible boundary conditions
- Zero boundary conditions (finite systems)
  - Useful especially in charged systems
- Periodic boundary conditions (bulk systems)

  Boundary conditions can be mixed
  - periodic in one dimension (wires)
  - periodic in two dimensions (surfaces)
Atomic orbital basis

Linear combination of atomic orbitals (LCAO) provide compact basis set

\[ \tilde{\psi}_n = \sum_{\nu} C_{n\nu} \Phi_{\nu}(r) \]

\[ \Phi_{n\ell m}^a(r) = R_{n\ell}^a \left( |r - R^a| \right) Y_{\ell m}(r - R^a) \]

The atomic orbitals are obtained from a free atom in a confining potential well

Systematic improvement of accuracy is non-trivial

Possible to switch between localized basis and real-space grids
Functions which are periodic with respect to unit cell can be written as sum of plane waves

\[ \psi_{nk}(r) = \frac{1}{\Omega^{1/2}} \sum_{G} C_{G,nk} e^{i(G+k) \cdot r} \]

\[ n(r) = \sum_{G} n_G e^{iG \cdot r} \quad \text{V}(r) = \sum_{G} V_G e^{iG \cdot r} \]

The expansion is truncated according to

\[ \frac{1}{2} |G + k|^2 < E_{\text{cutoff}} \]

Only periodic cells (supercells)
Comparison of basis sets

Real-space grids
- systematic convergence with single parameter
- good parallelization prospects
- some integrals complicated in real-space

Localized basis set
- compact basis
- systematic convergence can be difficult

Plane waves
- systematic convergence with single parameter
- some integrals simplified in reciprocal space
- very efficient in small to medium size systems
- parallelization more limited due FFTs
Using the Atomic Simulation Environment
Atomic Simulation Environment

ASE is a Python package for
- building atomic structures
- structure optimization and molecular dynamics
- analysis and visualization

ASE relies on external software which provides total energies, forces, etc.
- GPAW, Abinit, Siesta, Vasp, Castep, ...

Input files are Python scripts
- calculations are run as “python input.py”
- simple format, no knowledge of Python required
- knowledge of Python enables great flexibility

Simple graphical user interface

wiki.fysik.dtu.dk/ase
Setting up the atoms

Specifying atomic positions directly

```python
from ase.all import *  # Setup the atomic simulation environment

d0 = 1.10
x = d0 / sqrt(3)

atoms = Atoms('CH4',
               positions=[(0.0, 0.0, 0.0),  # C
                          (x, x, x),      # H1
                          (-x, -x, x),    # H2
                          (-x, x, -x),    # H3
                          (x, -x, -x)],   # H4
              )

view(atoms)
```

Reading atomic positions from a file

```python
...:
atoms = read('CH4.xyz')
view(atoms)
```

- Several file formats supported
Setting up the unit cell

By default, the simulation cell of an Atoms object has zero boundary conditions and edge length of 1 Å

Unit cell can be set when constructing Atoms

```
atoms = Atoms(..., # positions must be now in absolute coordinates
cell=(1., 2., 3.), pbc=True)# or pbc=(True, True, True)
```

or later on

```
atoms = Atoms(...) # positions in relative coordinates
atoms.set_cell((2.5, 2.5, 2.5), scale_atoms=True)
atoms.set_pbc(True) # or atoms.set_pbc((True, True, True))
```

```
atoms = ...
atoms.center(vacuum=3.5) # finite system 3.5 Å empty space around atoms
```

```
atoms = ...
atoms.set_pbc((False, True, True)) # surface slab
atoms.center(axis=0, vacuum=3.5) # 3.5 Å empty space in x-direction
```
Units in ASE

- Length: Å
- Energy: eV

Easy conversion between units:

```python
from ase.units import Bohr, Hartree

a = a0 * Bohr  # a0 in a.u., a in Å
E = E0 * Hartree  # E0 in Hartree, E in eV
```

- also Rydberg, kcal, nm, ...
Pre-defined molecules and structures

Database of small molecules (G2-1 and G2-2 sets)

```python
from ase.structure import molecule

mol = molecule('C6H6')  # coordinates from MP2 calculation
mol.center(3.5)  # molecule() returns unit cell of 1 Å
```

Bulk structures of elemental materials

```python
from ase.lattice import bulk

atoms = bulk('Si')  # primitive (2-atom) unit cell with exp. lattice constant
atoms_conv = bulk('Si', cubic=True)  # cubic 8-atom unit cell
atoms_my_a = bulk('Si', a=5.4)  # User specified lattice constant
```
Supercells and surfaces

Existing Atoms objects can be “repeated” and individual atoms removed

```python
from ase.lattice import bulk

atoms = bulk('Si', cubic=True)  # cubic 8-atom unit cell
supercell = atoms.repeat((4, 4, 4))  # 512 atom supercell
del supercell[0]  # remove first atom, e.g. create a vacancy
```

Utilities for working with surfaces

```python
from ase.lattice.surface import fcc111, add_adsorbate

slab = fcc111('Cu', size=(3,3,5))  # 5-layers of 3x3 Cu (111) surface
# add O atom 2.5 Å above the surface in the 'bridge' site
add_adsorbate(slab, 'O', 2.5, position='bridge')
```
Performing a calculation

In order to do calculation, one has to define a calculator object and attach that to Atoms

```python
from ase.structure import molecule          # Setup the atomic simulation environment
from gpaw import GPAW                   # Setup GPAW

atoms = molecule('CH4')
atoms.center(3.5)
calc = GPAW()                             # Use default parameters
atoms.set_calculator(calc)
atoms.get_potential_energy()           # Calculate the total energy
```

Specifying calculator parameters

```python
... 
calc = GPAW(h=0.18, nbands=6,              # 6 bands and grid spacing of 0.20 Å
            kpts=(4,4,4),                # 4x4x4 Monkhorst-Pack k-mesh
            xc='PBE', txt='out.txt')    # PBE and print text output to file
... 
```

See wiki.fysik.dtu.dk/gpaw/documentation/manual.html for all parameters
Performing a calculation

Serial calculations and analysis can be carried out with normal Python interpreter

```
[jenkovaa@flamingo ~]$ python input.py
```

Parallel calculations with `gpaw-python` executable

```
#PBS -N gpaw_test
#PBS -l select=4
#PBS -l walltime=00:20:00
...
aprun -n 96 gpaw-python input.py
```
from ase.all import *  # Setup the atomic simulation environment
from gpaw import GPAW  # Setup GPAW

atoms = ...
calc = GPAW(...)
atoms.set_calculator(calc)

opt = BFGS(atoms, trajectory='file.traj')  # define an optimizer
opt.run(fmax=0.05)  # optimize the structure until forces smaller than 0.05 eV / Å

See [wiki.fysik.dtu.dk/ase/ase/optimize.html](wiki.fysik.dtu.dk/ase/ase/optimize.html) for supported optimizers

“Best” optimizer is case-dependent
```python
import numpy as np

atoms = ...
calc = GPAW(...)  # Change the style of declaring an object.
atoms.set_calculator(calc)

# Check convergence with grid spacing
for h in [0.35, 0.30, 0.25, 0.20, 0.18]:
    txtfile = 'test_h' + str(h) + '.txt'
    calc.set(h=h, txt=txtfile)
    e = atoms.get_potential_energy()
    print(h, e)

# Change the style of declaring an object.
import numpy as np

atoms = ...
calc = GPAW(...)
atoms.set_calculator(calc)

# lattice constant for different XC-functionals
for xc in ['LDA', 'PBE']:
    for a in np.linspace(3.8, 4.3, 5):
        txtfile = 'test_xc_' + xc + '_a' + str(s) + '.txt'
        atoms.set_cell((a, a, a), scale_atoms=True)
        calc.set(xc=xc, txt=txtfile)
        e = atoms.get_potential_energy()
```
Saving and restarting

Saving full state of calculation: .gpw-files (or .hdf5-files)

```python
...
calc = GPAW(...) 
atoms.set_calculator(calc) 
atoms.get_potential_energy() # Calculate the total energy 
calc.write('myfile.gpw') # Atomic positions, densities, calculator parameters

...
calc.write('myfile.gpw', mode='all') # Save also wave functions (larger files)

...
calc.write('myfile.hdf5', mode='all') # If GPAW is build with HDF5 support
```

Restarting

```python
from ase.all import * # Setup the atomic simulation environment 
from gpaw import restart # Setup GPAW

atoms, calc = restart('file.gpw') 
e0 = atoms.get_potential_energy() # no calculation needed 
calc.set(h=0.20) 
e1 = atoms.get_potential_energy() # calculation total energy with new grid
```
Saving and restarting

Trajectories: atomic positions, energies, forces

```python
...  
calc = GPAW(...)  
atoms.set_calculator(calc)  
traj = PickleTrajectory('file.traj', 'w', atoms)  # define a trajectory file  
for a in np.linspace(3.8, 4.3, 5):  
    txtfile = 'test_xc_' + xc + '_a' + str(s) + '.txt'  
    atoms.set_cell((a, a, a), scale_atoms=True)  
    atoms.get_potential_energy()  
    traj.write()  # write cell and energy to trajectory file
```

Reading atomic positions

```python
from ase.all import *  # Setup the atomic simulation environment  
from gpaw import GPAW  # Setup GPAW  
  
atoms = read('file.traj')  # read the last image  
first = read('file.traj', 0)  # first image  
calc = GPAW(...)  
atoms.set_calculator(calc)  # calculator has to be attached
```
Simple graphical interface (ase-gui)

Trajectory can be investigated with **ase-gui** tool

```
[jenkovaa@flamingo ~]$ ase-gui file.traj
```

Investigate how total energy, forces, bond lengths etc. vary during simulation
Exercises

Go to wiki.fysik.dtu.dk/gpaw/exercises/exercises.html and get your hands dirty!
Time-dependent DFT with GPAW
Time-dependent DFT

- Generalization of density-functional theory also to time-dependent cases
  - one-to-one mapping between the time-dependent potential and the density
- Excited state properties
  - excitation energies, optical spectra, ...
- Time-dependent Kohn-Sham equations

\[ i \frac{\partial \psi_i(r, t)}{\partial t} = H \psi_i(r, t) \]

\[ H = -\frac{\nabla^2}{2} + V_{\text{eff}}(r, t) \]

\[ n(r, t) = \sum_i |\psi_i(r, t)|^2 \]
Real-time propagation

Direct integration of time-dependent Kohn-Sham equations in time-domain

\[ i \frac{\partial \psi_i(r, t)}{\partial t} = H(t) \psi_i(r, t) \]

\[ n(r, t) = \sum_i |\psi_i(r, t)|^2 \]

Integration is done with Crank-Nicholson type scheme:

\[ [1 + iH(t)\Delta t] \psi_i(t + \Delta t) = [1 - iH(t)\Delta t] \psi_i(t) \]

Initial value problem, the starting value \( \psi_i(t = 0) \) is obtained from ground state calculation
Optical absorption spectra from real-time TDDFT

- Excite the system with delta pulse \( E(t) = \epsilon k^0 \delta(t) \)
- Wave functions change instantaneously to
  \[
  \psi(t = 0^+) = \exp(\im \epsilon k^0 \cdot r) \psi(t = 0)
  \]
- Time-propagate wave functions and record the time-dependent dipole moment
- Spectra can be obtained via Fourier transform of the time-dependent dipole-moment
Practice

Perform ground state calculation and save the wave functions

```python
from ase.all import *
from gpaw import GPAW

atoms = ...
atoms.center(5.0)  # may need to be larger in real calculations
calc = GPAW(h=0.30)
atoms.set_calculator(calc)
atoms.get_potential_energy()
calc.write('gs.hdf5', mode='all')
```

Time-propagate

```python
from gpaw.tddft import TDDFT

time_step = 16.0  # as
iters = 650  # 650 x 16 as 10.4 fs
kick = [0,0,1e-3]  # Weak delta kick to z-direction

td_calc = TDDFT('gs.hdf5')
td_calc.absorption_kick(kick)
td_calc.propagate(time_step, iters, 'dmz.dat')
```
Calculate the optical spectra

```python
from gpaw.tddft import photoabsorption_spectrum

photoabsorption_spectrum('dmz.dat', 'spectrum_z.dat', width=0.2)
```
Practical details

- Depending on the symmetry of system, separate kick in all x,y,z directions may be needed.
- Normally, the size of simulation box has to be larger than in ground state calculations.
- Grid spacing can often be larger than in ground state calculations.
- Time-step affects the accuracy of spectra (too large time-step may lead into unstable propagation).
- Total simulation time affects the resolution of spectra.
Box size

![Graph showing the effect of different box sizes on dipole strength vs photon energy (hv) for various vacuum distances (R_{vac}). The graph compares R_{vac} = 7.2 Å, R_{vac} = 8.2 Å, and R_{vac} = 11.2 Å. The text notes that a small box is NOT OK, while a larger box damps artificial oscillations.](image)

- Small box OK
- Ionization threshold
- Larger box damps artificial oscillations
- Small box NOT OK
Grid spacing

Eigenvalues are important, not forces or total energy!
Time step

UV/VIS

Higher energy => larger error

Dipole strength [1/eV]

hv [eV]
Total simulation time

Spectrum is convoluted by sinc-function \((=\sin(x)/x)\)

\[
\int_0^T \mu(t)e^{-i\omega t}dt = \int_{-\infty}^{\infty} f(\omega) \frac{\sin(T\omega)}{T\omega} d\omega
\]

Double simulation time => double resolution
Linear response TD-DFT

Small perturbation to ground state potential

\[ V_{\text{ext}} = V_{\text{ext},0} + \delta V_{\text{ext}}(t) \]

results a change in density

\[ n(r, t) = n_0(r) + n_1(r, t) \]

Within linear response:

\[ n_1(r, \omega) = \int d^3r \chi(r', \omega) \delta V_{\text{ext}}(r', \omega) \]

TD-DFT:

\[ n_1(r, \omega) = \int d^3r \chi_0(r, r', \omega) \delta V_{\text{eff}}(r', \omega) \]

The interacting response function can be obtained from the non-interacting one
Response function

Dyson-like equation for density-density response:

\[ \chi(r, r', \omega) = \chi_0(r, r', \omega) \]
\[ + \iint_\Omega dr_1 dr_2 \chi_0(r, r_1, \omega) K(r_1, r_2) \chi(r_2, r', \omega) \]

Non-interacting response function is constructed from ground state Kohn-Sham orbitals

Coupling kernel is:

\[ K(r_1, r_2) = \frac{1}{|r_1 - r_2|} + f_{xc}[n](r_1, r_2) \]

with XC kernel

\[ f_{xc}(r_1, r_2) = \frac{\partial v_{xc}[n](r_1)}{\partial n(r_2)} \]
Response function

Non-interacting response function is given by

\[ \chi^0(r, r', \omega) = \sum_{k,q} \sum_{n,n'} BZ \frac{f_{nk} - f_{n'k+q}}{\omega + \epsilon_{nk} - \epsilon_{n'k+q} + i\eta} \psi^*_n(r) \psi_{n'k+q}(r) \psi_n(r') \psi^*_{n'k+q}(r') \]

Occupation numbers, eigenvalues and ground state orbitals of occupied and unoccupied states

In extended systems one solves the Dyson equation in plane wave basis

- ground state calculation can be done in any basis
Dielectric matrix

Dielectric matrix is related to the interacting response function

$$\epsilon_{GG'}^{-1}(q, \omega) = \delta_{GG'} + \frac{4\pi}{|q + G|^2} \chi_{GG'}(q, \omega)$$

Macroscopic dielectric function

$$\epsilon_M(q, \omega) = \frac{1}{\epsilon_{00}^{-1}(q, \omega)}$$

Optical spectra

$$\text{ABS} = \text{Im} \epsilon_M(q \to 0, \omega)$$

Electron-energy loss spectra

$$\text{EELS} = -\text{Im} \frac{1}{\epsilon_M(q, \omega)}$$
Excitation energies in finite systems

Excitation energies can be calculated from eigenvalue equation:

\[ \Omega F_l = \omega_l^2 F_l, \]

where

\[ \Omega_{ij\sigma,kl\tau} = \delta_{ik} \delta_{j\ell} \delta_{\sigma\tau} \varepsilon_{ij\sigma}^2 + 2 \sqrt{\varepsilon_{ij\sigma} \varepsilon_{kl\tau}} K_{ij\sigma,kl\tau}, \]

\[ \varepsilon_{ij} = \varepsilon_i - \varepsilon_j \]

with the coupling kernel

\[ K_{ij\sigma,kl\tau} = \int dr_1 dr_2 n^*_{ij\sigma}(r_1) \left[ \frac{1}{|r_1 - r_2|} + f_{xc}(r_1, r_2) \right] n_{kl\tau}(r_2) \]

\[ n_{ij\sigma}(r) = \psi_{i\sigma}(r)^* \psi_{j\sigma}(r) \]

i and j indexes go through occupied and unoccupied states, respectively.
Optical spectra in finite systems

Dipole oscillator strengths of excitations can be calculated from the eigenvectors

\[ f_{I\alpha} = \left| \sum_{ij\sigma} \langle \psi_{i\sigma} | r_{\alpha} | \psi_{j\sigma} \rangle \sqrt{\varepsilon_{ij\sigma}} F_I \right|^2 \]

Spectra with finite peak widths are obtained by folding the oscillators strengths e.g. with Gaussian

\[ S_{\alpha}(\omega) = \sum_I f_{I\alpha} g(\omega - \omega_I) \]
Practice

Perform ground state calculation and include also unoccupied states

```python
from ase.all import *  # Setup the atomic simulation environment
from gpaw import GPAW  # Setup GPAW
from gpaw.eigensolvers import CG  # Conjugate gradient eigensolver

atoms = ...
atoms.center(vacuum=5.0)  # More vacuum might be needed in reality...
calc = GPAW(nbands=1, h=0.30, txt='Na2_gs.txt')
atoms.set_calculator(calc)
e = atoms.get_potential_energy()

# Calculate also unoccupied states with the fixed density
eig = CG()  # unoccupied states converge often better with cg
calc.set(nbands=20, convergence={'bands': 'all'},  # converge unoccupied states
         eigensolver=eig,
         fixdensity=True)
e = atoms.get_potential_energy()
# write the wave functions to a file
calc.write('na2_gs.gpw', 'all')
```
Practice

Calculate $\Omega$ matrix

```python
from gpaw import *  # Setup GPAW
from gpaw.lrtddft import *

atoms, calc = restart('na2_gs.gpw')  # read in a ground state calculation

# Calculate the omega matrix
lr = LrTDDFT(calc, xc='LDA')
# Save the omega matrix
lr.write('Omega_Na2.gz')
```

Diagonalization can be performed in separate step

```python
from gpaw.lrtddft import *

# Read the omega matrix from a file
lr = LrTDDFT(filename='Omega_Na2.gz')
# Diagonalize the matrix
lr.diagonalize()
# Print out five lowest excitations
lr.analyse(range(5))
# Calculate the absorption spectrum and save it to a file
photoabsorption_spctrum(lr, 'Na2_spectrum.dat', e_min=0.0, e_max=10, width=0.1)
```
Practical details

In addition to the box size and grid spacing, accuracy is controlled by the number of electron-hole pairs.

The computational intensity of the calculation is $O(N_{\text{eh}}^3)$.

The size of electron-hole basis can be reduced:

```python
lr = LrTDDFT(calc, xc='LDA',
           istart=5, # first occupied state to consider
           jend=20) # last unoccupied state to consider
```

The proper number of eh-pairs is very system dependent.
Real-time vs. linear response

Real-time
- only excitations corresponding to given perturbation
- non-linear effects
- scales $O(N^2)$ with the system size, large prefactor
- time step controls the accuracy relatively straightforwardly

Linear response
- all excitations (within linear response)
- scales $O(N^6)$ with the system size, small prefactor
- control of accuracy by the electron-hole basis size can be complex
Parallel calculations with GPAW
Parallelization levels

$$H_{k,s}\psi_{i,k,s}(r) = e_i S\psi_{i,k,s}(r)$$

$$i\partial_t \psi_{i,s}(r,t) = H_s \psi_{i,s}(r,t)$$

$$n(r) = \sum_{i,k,s} |\psi_{i,k,s}(r)|^2$$

- Parallelization over all degrees of freedom
  - real-space grid
  - \textit{k-points} and spin
  - electronic states

- Additional trivial parallelizations possible
  - Electron-hole pairs
  - different atomic configurations or unit cells
Parallelization over k-points and spin

$H_{k,s} \psi_{i,k,s}(r) = e_i \psi_{i,k,s}(r)$

$n(r) = \sum_{i,k,s} |\psi_{i,k,s}(r)|^2$

- Spin and k-points are treated equivalently
- Trivial parallelization
- Limited scalability
  - k-points only in (small) periodic systems
  - spin only in magnetic systems
- All basis sets
Parallelization over real-space grid

$$H_{k,s} \psi_{i,k,s}(r) = E_i \psi_{i,k,s}(r)$$

$$n(r) = \sum_{i,k,s} |\psi_{i,k,s}(r)|^2$$

- Domain decomposition

Finite difference Laplacian
PAW augmentation sphere

- Only local communication
- Good parallel scalability down to domain sizes $\sim 16 \times 16 \times 16$
- Not available in plane wave mode
Parallelization over electronic states

\[ H_{k,s} \psi_{i,k,s}(r) = e_i \psi_{i,k,s}(r) \]

\[ n(r) = \sum_{i,k,s} |\psi_{i,k,s}(r)|^2 \]

- Nearly trivial parallelization in real-time TDDFT
  - (similar to k-points)
  - good scalability down to 20 states per process

- Orthonormalizations are complicated in ground state DFT
  - communication of all wave functions to all processes
  - parallel scalability down to 150-250 states per process
  - all basis sets
Parallelization over electron-hole pairs

*Casida equation in linear response TD-DFT:*

\[ \Omega F_1 = \omega^2_1 F_1, \]

\[ \Omega_{ij\sigma,kl\tau} = \delta_{ik}\delta_{jl}\delta_{\sigma\tau}\varepsilon^2_{ij\sigma} + 2\sqrt{\varepsilon_{ij\sigma}\varepsilon_{kl\tau}} K_{ij\sigma,kl\tau} \]

Matrix elements can be calculated independently

Nearly trivial parallelization over electron-hole pairs \( ij \)

Domain decomposition for individual matrix elements
Parallel scalability

Ground state DFT
- 561 Au atom cluster
- ~6200 electronic states
- Blue Gene P, Argonne
Parallel scalability

- Real-time TD-DFT
  - 702 Si atom cluster
  - ~2800 electronic states
  - Cray XT5 Jaguar, Oak Ridge
Parallel scalability

Linear-response TD-DFT
- $\text{Au}38(\text{SCH3})24$ cluster $\sim160$ atoms
- $\sim680$ electronic-hole pairs
- Cray XE6 Hermit, HLRS, Germany
Parallel performance

Ground state DFT
- Large MgH2 cell, 1296 atoms
- Cray XC30, CSC (node=16 cores)
Practical details

- k-points have to be distributed evenly
  - same number of k-points in each process

- Electronic states have to be distributed evenly
  - same number of states in each process

- In principle, arbitrary number of processes can be used for domain decomposition
  - recommended to use cubic domains e.g. 4 x 4 x 4
  - recommended to use domain dimensions which factor the number of grid points
Practical details

By default, k-point and spin are distributed first, and the remaining processors are used for domain decomposition.

Example: magnetic system, 5 k-points
- with 20 processors: 10 (=2x5) processors for spin/k-point and 2 processors for domain decomposition (2x1x1 layout)
- with 24 processors: 2 processors for spin, 12 processors for domain decomposition (3x2x2 layout)
Practical details

Electronic state parallelization has to be specified explicitly

```aprungpaw-python input.py --state-parallelization=2```

- 2 processors for states, 256 to k-points/spin/domains

For large calculations (> 1000 states) one more command line argument:

```aprun -n 512 gpaw-python input.py --sl_default=4,4,64```

- Some large matrix diagonalizations are done in parallel with 16 (=4x4) processes
- 4x4 or 8x8 are typically good values (block size 64 has only small effect)
Parallelization options can be given also as GPAW calculator parameters

```python
... calc = GPAW(...
    parallel={
      kpt : 4,  # k-point parallelization with 4 MPI tasks
      domain : 8,  # domain decomposition with 8 MPI tasks
      band : 2,  # band parallelization with 2 MPI tasks
      s1_default : (4,4,64)}  # parallel matrix diagonalization

...)
```

Command line arguments precede calculator parameters
“dry-run” mode

Often, it is desirable to check system parameters without an actual calculation.

GPAW offers a dry-run mode:

- only unexpensive initializations, can be run serially
- simulates parallel calculations and shows the parallelization scheme
- estimates the memory usage

[cscuser@cobol ~]$ python input.py --state-parallelization=2 --dry-run=512