koopmans

Release v1.0.1

Edward Linscott, Riccardo De Gennaro and Nicola Colonna

Dec 14, 2023

CONTENTS

Fig. 1: a package for performing and automating Koopmans functional calculations

CHAPTER

ONE

ABOUT

koopmans is a package for performing Koopmans spectral functional calculations with Quantum ESPRESSO, developed by researchers in the THEOS research group at EPFL.

The package includes…

- the official version of Quantum ESPRESSO, which contains Koopmans functionals implemented with a k-point framework
- a modified version of Quantum ESPRESSO v4.1 with Koopmans functionals implemented within the cp.x code (Γ-only)
- koopmans, a python package and executable that makes running Koopmans spectral calculations with the above codes easy!

CHAPTER

TWO

QUICK START

1. install the code following the *installation instructions*
2. download the `input file for tutorial 1`
3. run `koopmans ozone.json`

Having done this, you have run your very first Koopmans calculation! You will see that a number of files have been generated. These contain the outputs of the calculation. For example, the ionisation potential and electron affinity of ozone are given by the negative of the HOMO and LUMO listed in the file `final/ki_final.cpo`. For more details, see *tutorial 1*.

CHAPTER

THREE

THEORY

This section contains a brief introduction to Koopmans functionals.

3.1 Quasiparticle energies, piecewise linearity, and Koopmans' theorem

Density functional theory (DFT) is a remarkably successful theory. However, its prediction of quasiparticle energies is very unreliable. Indeed, while the Kohn-Sham eigenvalues may loosely mirror experimental quasiparticle energies, there is formally no connection between the two (except for the HOMO in exact DFT, which is related to the decay of the density at large distances).

Furthermore, because DFT is an approximate theory the Kohn-Sham orbitals suffer from a few well-known errors, making them an even worse proxy for quasiparticle energies. Chief among these errors is "self-interaction error" (SIE).

DFT suffers from "one-body self-interaction error" because of the way it treats the Hartree term. For a wavefunction represented by a single Slater determinant, the Hartree term Φ_i felt by the i$^{\text{th}}$ particle is given by

$$\Phi_i(\mathbf{r}) = \sum_{i \neq j} \int d\mathbf{r}' \frac{|\psi_{n_j}(\mathbf{r}')|^2}{|\mathbf{r} - \mathbf{r}'|}$$

but in DFT we replace this orbital-dependent term with

$$\Phi(\mathbf{r}) = \int d\mathbf{r}' \frac{\rho(\mathbf{r})}{\mathbf{r} - \mathbf{r}'}$$

which ignores the $i \neq j$ of the sum. This would be perfectly fine if the xc-functional perfectly cancelled this self-Hartree term, but most xc-functionals do not. Consequently, KS particles tend to over-delocalise in order to minimise the Coulomb repulsion they feel from their own density.

More generally, DFT suffers from "many-body self-interaction error" (or "delocalisation error"). This manifests itself as an erroneous curvature in the total energy $E(N)$ of the system as a function of the total number of electrons N. Compare this to the exact functional, which we know should be piecewise-linear between the energy at integer occupancies.

Fig. 1: $E(N)$ for the exact functional, semi-local DFT, and Hartree-Fock

This erroneous curvature directly impacts the Kohn-Sham eigenvalues. For instance, consider the energy of the highest occupied molecular orbital (HOMO), which is given by

$$\varepsilon_{HO} = \left.\frac{\partial E}{\partial N}\right|_{N=N^-}$$

that is, the gradient of $E(N)$ approaching N from the left. In principle, this energy should be equal to the (indeed we can see this for the exact functional, where the gradient is given by

$$\varepsilon_{HO}^{\text{exact}} = \left.\frac{\partial E^{\text{exact}}}{\partial N}\right|_{N=N^-} = E^{\text{exact}}(N) - E^{\text{exact}}(N-1)$$

However, we can see in the following figure that due to the erroneous curvature in the semi-local functional

$$\varepsilon_{HO}^{\text{sl}} = \left.\frac{\partial E^{\text{sl}}}{\partial N}\right|_{N=N^-} > E^{\text{sl}}(N) - E^{\text{sl}}(N-1)$$

Fig. 2: Close-up of $E(N)$ for semi-local DFT, showing the consequences of SIE for quasiparticle energies

That is, the Kohn-Sham HOMO eigenvalue is overestimated due to the presence of SIE.

It is not just the HOMO energy that is affected by SIE. By similar logic we can show that SIE affects all of the Kohn-Sham eigenvalues, and by extension it will detrimentally affect spectral properties such as densities of states, band structures, and optical spectra.

Given these failures of semi-local DFT, the question becomes how can we relate Kohn-Sham eigenvalues to quasiparticles while addressing self-interaction? The answer: Koopmans functionals.

TODO discuss Koopmans theorem

3.2 Koopmans functionals

3.2.1 The motivating idea behind the functionals

We saw from the previous section that DFT Kohn-Sham eigenvalues…

a. are not formally related to quasiparticle energies
b. suffer from self-interaction error

These two points inspire the design of Koopmans functionals. The key idea behind these functionals is that we desire a functional whose orbital[1] energies

$$\varepsilon_i^{\text{Koopmans}} = \langle \varphi_i | H | \varphi_i \rangle = \frac{dE_{\text{Koopmans}}}{df_i}$$

possess two key properties:

1. $\varepsilon_i^{\text{Koopmans}}$ is independent of the corresponding orbital occupancy f_i
2. $\varepsilon_i^{\text{Koopmans}}$ is equal to the corresponding DFT total energy difference $\Delta E_i^{\text{Koopmans}}$ that corresponds to the addition/removal of an electron from the corresponding orbital i

Property 1 means that we will be SIE-free, because the curvature of E^{Koopmans} with respect to f_i will be zero for every orbital i. Meanwhile, property 2 is a necessary condition for piecewise linearity of the total energy. It also means that the quasiparticle energies are on much more stable theoretical footing, because they are expressly related to total energy differences, which are ground-state properties.

[1] To be specific, by "orbitals" we mean the *variational* orbitals (explained *here*)

3.2.2 Derivation of the functionals

So how do we construct a functional that posesses these properties? The brief derivation is as follows: let us assume a functional of the form

$$E^{\text{Koopmans}} = E^{DFT} + \sum_i \Pi_i$$

This is a "corrective" functional – that is, we start from the exact or an approximate DFT energy functional E^{DFT} (the "base" functional) and add some as-of-yet undetermined corrections Π_i to each orbital (indexed by i). If we take the derivative with respect to the occupancy of the j^{th} orbital then we have

$$\eta_j = \frac{dE^{DFT}}{df_j}\bigg|_{f_j=s} + \frac{d\Pi_j}{df_j}\bigg|_{f_j=s} = \langle\varphi_j|\hat{h}^{\text{DFT}}(s)|\varphi_j\rangle + \frac{d\Pi_j}{df_j}\bigg|_{f_j=s}$$

where we assumed that the cross-term derivatives $d\Pi_i/df_j$ vanish, and because E^{Koopmans} ought to be linear in f_j, we replaced its derivative with some yet-to-be deterimined constant η_j. For the second equality we invoked Janak's theorem.

Assuming that the our energy correction Π_j is zero at integer occupancies, and neglecting for the moment any orbital relaxation as the orbital occupancies change, it follows that

$$\Pi^u_j = -\int_0^{f_j} \langle\varphi_j|\hat{h}^{\text{DFT}}(s)|\varphi_j\rangle ds + f_j\eta_j$$

where the u superscript denotes the fact that we neglected orbital relaxation, and thus this term is "unscreened". To account for this screening we must introduce some screening parameters $\{\alpha_i\}$ such that $\Pi_j = \alpha_j\Pi^u_j$.

Thus we arrive at the form of the Koopmans functional:

$$E^{\text{Koopmans}}[\rho,\{f_i\},\{\alpha_i\}] = E^{DFT}[\rho] + \sum_i \alpha_i \left(-\int_0^{f_i}\langle\varphi_i|\hat{h}^{\text{DFT}}(s)|\varphi_i\rangle ds + f_i\eta_i\right)$$

This functional, by construction, has orbital energies that possess two key properties discussed *above*.

3.2.3 Understanding the functional

To understand this functional a little better, consider the following figure, that represents the total energy of a system with $N + f_i$ electrons, with a partially-occupied frontier orbital i.

Fig. 3: Illustration of the Koopmans functional, term-by-term

What energy does the Koopmans functional assign to this system? Well, it starts with the energy given by the base functional $E^{DFT}[\rho]$. This is denoted as point 1 in the above figure. From this term we subtract $\alpha_i \int_0^{f_i}\langle\varphi_i|\hat{h}^{\text{DFT}}(s)|\varphi_i\rangle ds$ – that is, we remove the spurious non-linear dependence of the energy on the orbital occupancy. This takes us from point 1 to point 2 in the figure. Finally we add back in $\alpha_i f_i\eta_i$, a term that is linear in f_i with some as-of-yet unspecified gradient $\alpha_i\eta_i$ (point 3).

The end result is that E^{Koopmans} is explicitly linear in f_i and thus it satisfies property 1 by construction. As for property 2, the value of $\varepsilon_i^{\text{Koopmans}}$ is given by $\alpha_i\eta_i$. We will therefore choose our these two parameters in order to guarantee that these quasiparticle energies correspond to the desired total energy differences. We will discuss how exactly this is done *later*.

3.3 The key ingredients in a Koopmans calculation

3.3.1 The variational orbitals

The one important distinction that is worth making right away is that Koopmans functionals are not density functionals, but *orbital-density-dependent* (ODD) functionals. This is because in addition to being dependent on the total density ρ they are also dependent on the individual occupancies. Indeed, each orbital will be subjected to a different potential, and when we solve a Koopmans functional we must minimise the total energy with respect to the entire set of orbitals as opposed to just the total density.

A further complication of ODDFTs is that we actually have *two* sets of orbitals that we must be careful to distinguish. The set of orbitals that minimise the total energy are the so-called *variational* orbitals. Because the leading term in an orbital's Koopmans potential is the negative of that orbital's self-Hartree energy, these variational orbitals tend to be very localised.

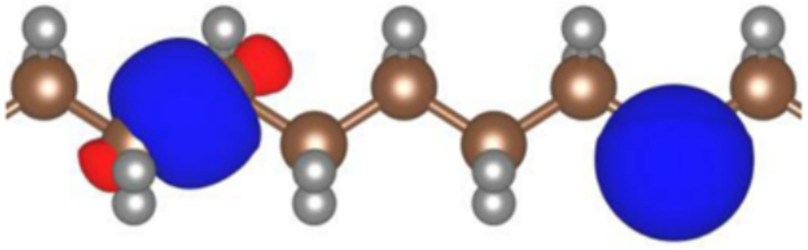


Fig. 4: Two variational orbitals of polyethylene. Figure taken from [24]

If we have minimised the total Koopmans energy we can then construct the Hamiltonian. If we then diagonalise this Hamiltonian we would obtain the so-called *canonical* orbitals. In a DFT framework, diagonalising the Hamiltonian would yield exactly the same orbitals that minimise the total energy. However, in an ODDFT, this is not the case, because the total energy is not invariant with respect to unitary rotations of a given set of orbitals, and thus the variational and canonical orbitals are different. In contrast to the variational orbitals, the canonical orbitals are typically very delocalised and much more closely resemble the Kohn-Sham orbitals of DFT.

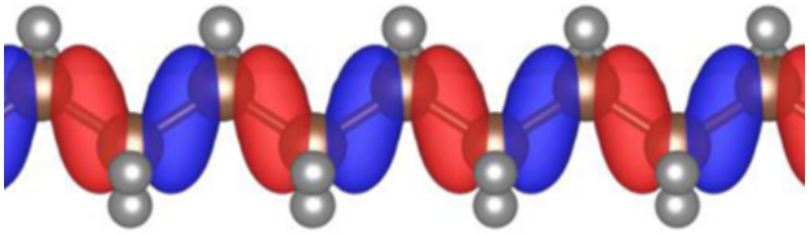


Fig. 5: A canonical orbital of polyethylene. Figure taken from [24]

Note: Transforming a DFT to an ODDFT may seem like a bothersome complication but actually it is a natural generalization – indeed, an ODDFT is in fact an energy-discretized spectral functional theory [12].

3.3.2 The screening parameters

In any Koopmans calculation, we must obtain the set of screening parameters $\{\alpha_i\}$. As we discussed earlier, we would like the functional's total energy to be piecewise linear i.e. we would like quasiparticle energy to match the corresponding total energy differences.

Specifically, we would like $\varepsilon_i^{\text{Koopmans}} = \Delta E_i^{\text{Koopmans}}$, where

$$\Delta E_i^{\text{Koopmans}} = \begin{cases} E^{\text{Koopmans}}(N) - E_i^{\text{Koopmans}}(N-1) & \text{filled orbitals} \\ E_i^{\text{Koopmans}}(N+1) - E^{\text{Koopmans}}(N) & \text{empty orbitals} \end{cases}$$

where $E_i^{\text{Koopmans}}(N \pm 1)$ is the total energy of the system where we add/remove an electron from variational orbital i and allow the rest of the system to relax.

We will use this condition to determine the screening parameters *ab initio*. In order to do so, there are two potential approaches: (a) via *SCF calculations* or (b) via *DFPT*.

SCF

In this approach, we explicitly calculate all of the energy differences $\Delta E_i^{\text{Koopmans}}$ via a series of constrained Koopmans or DFT calculations. Specifically, given a starting guess $\{\alpha_i^0\}$ for the screening parameters, an improved guess for the screening parameters can be obtained via

$$\alpha_i^{n+1} = \alpha_i^n \frac{\Delta E_i - \varepsilon_i^0(1)}{\varepsilon_i^{\alpha_i^n}(1) - \varepsilon_i^0(1)}$$

for filled orbitals and

$$\alpha_i^{n+1} = \alpha_i^n \frac{\Delta E_i - \varepsilon_i^0(0)}{\varepsilon_i^{\alpha_i^n}(0) - \varepsilon_i^0(0)}$$

for empty orbitals, where

$$\varepsilon_i^{\alpha_i}(f) = \left.\frac{\partial E^{\text{Koopmans}}}{\partial f_i}\right|_{f_i=f} = \left.\langle\varphi_i|\hat{H}_{\text{DFT}} + \alpha_i \hat{v}_i^{\text{Koopmans}}|\varphi_i\rangle\right|_{f_i=f}$$

All of these quantities for calculating α_i^{n+1} are obtained from constrained Koopmans and DFT calculations. Specifically, a N-electron Koopmans calculation yields $E^{\text{Koopmans}}(N)$, $\varepsilon_i^{\alpha_i^n}$, and ε_i^0, and a constrained $N \pm 1$-electron calculation yields $E_i^{\text{Koopmans}}(N \pm 1)$.

Typically, very few iterations are required in order to reach self-consistency.

Note: For a periodic system, this method for determining the screening parameters requires a supercell treatment. This is because the $N \pm 1$-electron systems have a charged defect and a supercell is required in order to avoid spurious interactions between periodic images.

DFPT

While the SCF approach can provide us with all of the ingredients to calculate the screening parameters, it is a somewhat cumbersome approach. We must perform several constrained DFT/Koopmans calculations, and for periodic systems these must be performed in a supercell.

An alternative to the SCF approach is to take advantage of density functional perturbation theory (DFPT) [2] in order to compute the screening coefficients. In this approach the energy is approximated as a quadratic function of the occupation number, and the expression for the screening coefficients reduces to

$$\alpha_i = \frac{d^2 E_{\mathrm{DFT}}/df_i^2}{\partial^2 E_{\mathrm{DFT}}/\partial f_i^2} = \frac{\langle n_i|\epsilon^{-1} f_{\mathrm{Hxc}}|n_i\rangle}{\langle n_i|f_{\mathrm{Hxc}}|n_i\rangle}$$

where $\frac{d}{df_i}$ and $\frac{\partial}{\partial f_i}$ represent variations done accounting for the orbitals relaxation or not, respectively, $\epsilon(\mathbf{r},\mathbf{r}')$ is the microscopic dielectric function of the material, $f_{\mathrm{Hxc}}(\mathbf{r},\mathbf{r}') = \delta^2 E_{Hxc}/\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}')$ is the Hartree-exchange and correlation kernel, and $n_i(\mathbf{r}) = |\varphi_i(\mathbf{r})|^2$ is the orbital density.

The evaluation of the screening coefficients within this approach only requires quantities available from a N-electron calculation. Specifically, we can rewrite the above equation in terms of the density response $\Delta^i\rho$ to a perturbing potential V^i_{pert} generated by the orbital density n_i:

$$\alpha_i = 1 + \frac{\langle V^i_{\mathrm{pert}}|\Delta^i\rho\rangle}{\langle n_i|V^i_{\mathrm{pert}}\rangle}$$

The advantage of this approach compared to the SCF is that there is no need for a supercell treatment, and in the case of periodic solids a primitive cell implementation can be used. By exploiting Bloch symmetries the linear response formula for the screening coefficients can be decomposed into a set of independent problems, one for each q point sampling the Brillouin zone of the primitive cell

$$\alpha_i = 1 + \frac{\sum_{\mathbf{q}}\langle V^i_{\mathrm{pert},\mathbf{q}}|\Delta^i_{\mathbf{q}}\rho\rangle}{\sum_{\mathbf{q}}\langle \rho^i_{\mathbf{q}}|V^i_{\mathrm{pert},\mathbf{q}}\rangle}$$

This greatly reduces the computational cost of the calculation (see below), but as a consequence the DFPT approach has a few limitations:

1. We have approximated the energy to second order. In most cases this is very accurate, correctly capturing the leading Hartree contribution and only missing higher-order xc contributions.
2. It is only currently implemented for the KI functional. For KIPZ, the PZ kernel (i.e. the second derivative of the PZ energy with respect to the density) is required and this is not implemented in common electronic structure codes. (See *below* for an explanation of what we mean by "KI" and "KIPZ".)

For more details regarding the DFPT method, see [7].

Computational scaling for periodic systems

In the SCF approach, the screening coefficients are computed within a supercell and with a finite difference approach, by performing additional total-energy calculations where the occupation of a single variational orbital is constrained [24]. This requires an SCF calculation per orbital, which takes a computational time T^{SC} that roughly scales as $\mathcal{O}\left(\left(N_{\mathrm{el}}^{\mathrm{SC}}\right)^3\right)$, where $N_{\mathrm{el}}^{\mathrm{SC}}$ is the number of electrons in the supercell.

The DFPT approach scales as $T^{\mathrm{PC}} \propto N_{\mathbf{q}} N_{\mathbf{k}} {N_{\mathrm{el}}^{\mathrm{PC}}}^3$. This is the typical computational time for the SCF cycle $N_{\mathbf{k}} {N_{\mathrm{el}}^{\mathrm{PC}}}^3$, times the number of independent monochromatic perturbations $N_{\mathbf{q}}$.

Using the relation $N_{\mathrm{el}}^{\mathrm{SC}} = N_{\mathbf{k}} N_{\mathrm{el}}^{\mathrm{PC}}$, and the fact that $N_{\mathbf{q}} = N_{\mathbf{k}}$, the ratio between the supercell and primitive computational times is $T^{\mathrm{SC}}/T^{\mathrm{PC}} \propto N_{\mathbf{q}}$. Therefore as the supercell size (and, equivalently, the number of $\mathbf{q}$-points in the primitive cell) increases, the primitive-cell-DFPT approach becomes more and more computationally efficient.

3.3.3 The flavour: KI or KIPZ

As we have seen *previously*, there is some freedom in how we define our Koopmans functional. Namely, we need to choose values for η_i, the gradient of the energy as a function of the the occupancy of orbital i, for each value of i (modulo the corresponding screening term).

There are several different ways to define these gradient terms, and each approach gives rise to a different "flavour" of Koopmans functionals.

KI

In the "KI" approach, η_i is chosen as the total energy difference of two adjacent electronic configurations with integer occupations:

$$\eta_i^{\mathrm{KI}} = E^{\mathrm{DFT}}[f_i = 1] - E^{\mathrm{DFT}}[f_i = 0] = \int_0^1 \langle \varphi_i | \hat{h}^{\mathrm{DFT}}(s) | \varphi_i \rangle ds$$

in which case the explicit expression for the unscreened KI Koopmans' correction becomes

$$\Pi_i^{\mathrm{KI}} = f_i \eta_i^{\mathrm{KI}} = E_{\mathrm{Hxc}}[\rho - \rho_i] - E_{\mathrm{Hxc}}[\rho] + f_i \left(E_{\mathrm{Hxc}}[\rho - \rho_i + n_i] - E_{\mathrm{Hxc}}[\rho - \rho_i] \right)$$

where $\rho_i(\mathbf{r}) = f_i |\varphi_i(\mathbf{r})|^2$ and $n_i(\mathbf{r}) = |\varphi_i(\mathbf{r})|^2$. E_{Hxc} denotes the Hartree and exchange-correlation energy corresponding to the underlying base functional.

Note: It can be seen that at integer occupations, the KI energy correction vanishes i.e. $\Pi_i^{\mathrm{KI}} = 0$. In other words, the KI functional preserves the potential energy surface of the base functional! But while the energy is vanishing, the potential is non-vanishing, which means that the KI correction will affect the spectral properties of the system.

KIPZ

In the "KIPZ" approach, the slope η_i is also chosen as the total energy difference of two adjacent electronic configurations with integer occupations, but this time using the Perdew-Zunger (PZ) one-electron-self-interaction corrected (SIC) functional applied to the approximate DFT base functional:

$$\eta_i^{\mathrm{KIPZ}} = E^{\mathrm{PZ}}[f_i = 1] - E^{\mathrm{PZ}}[f_i = 0] = \int_0^1 \langle \varphi_i | \hat{h}_i^{\mathrm{PZ}}(s) | \varphi_i \rangle ds,$$

providing the explicit expression for the unscreened Π_i^{KIPZ} correction

$$\Pi_i^{\mathrm{KIPZ}} = -\int_0^{f_i} \langle \varphi_i | \hat{h}^{\mathrm{DFT}}(s) | \varphi_i \rangle ds + f_i \int_0^1 \langle \varphi_i | \hat{h}_i^{\mathrm{PZ}}(s) | \varphi_i \rangle ds \ .$$

where

$$\hat{h}_i^{\mathrm{PZ}}(s) = \hat{h}^{\mathrm{DFT}}(s) - \hat{v}_{\mathrm{Hxc}}^{\mathrm{DFT}} \left[s |\varphi_i(\mathbf{r})|^2 \right]$$

is the PZ self-interaction correction applied to the i^{th} variational orbital. This correction removes the Hartree-plus-exchange-correlation potential for that orbital.

This correction can be rewritten as

$$\Pi_i^{\mathrm{KIPZ}} = \Pi_i^{\mathrm{KI}} - f_i E_{\mathrm{Hxc}}[n_i]$$

Note: In the unscreened case ($\alpha_i = 1$) the KIPZ functional can be thought of as the KI correction applied to the PZ-SIC functional (this can be verified by replacing the base DFT functional and Hamiltonian with its PZ-SIC counterparts). However, in the general case of $\alpha_i \neq 1$ the KIPZ functional form implies also scaling each PZ self-interaction correction with its own screening coefficient.

For more details, refer to [3].

3.4 The Koopmans workflows

In a semi-local DFT calculation, all that one needs to do is obtain the ground state density. As discussed in the previous section, for Koopmans calculations we must also obtain the minimising variational orbitals as well as the screening parameters. This means that, compared to a standard semi-local DFT calculation, a few additional steps are required.

Typically, a Koopmans calculation can be divided into three stages

1. an initialization step, where the density and variational orbitals are initialized
2. the calculation of screening parameters
3. a final calculation using the obtained variational orbitals and screening parameters

Depending on the method for calculating screening parameters, the resulting workflow can look quite different. Differences also emerge between molecules and solids. For the latter, maximally localised Wannier functions are typically used as variational orbitals, which necessitates an additional Wannierization procedure, performed using Wannier90.

Fig. 6: Flowchart of the SCF workflow (click to enlarge)

Fig. 7: The DFPT workflow (click to enlarge)

As you can see, these workflows can become quite complicated, but do not worry! The `koopmans` code automates the process of running these workflows.

3.5 Limitations

3.5.1 Issues with metals

TODO

3.6 Related methods

Koopmans functionals have resonance with various methods developed by other groups. These include…

- the Wannier transition-state method of Anisimov and Kozhevnikov [1]
- the optimally-tuned range-separated hybrid functionals of Kronik, Neaton, and others [15, 31]
- the localized orbital scaled correction of Yang and others [17]
- the ensemble DFT method of Kreisler, Kronik and others [14]
- the Koopmans-Wannier method of Wang and others [19]

For more details, refer to our various *publications*.

CHAPTER

FOUR

INSTALLATION

4.1 Downloading

Koopmans is available to download on github. You can clone it directly with

```
git clone --recursive git@github.com:epfl-theos/koopmans.git
```

4.2 Installing

4.2.1 Quick installation

For a quick installation one can simply run `make; sudo make install`

4.2.2 Detailed installation

Setting up a virtual environment

You are encouraged (but it is not necessary) to first create and activate a virtual environment as follows:

```
sudo apt-get install python3-pip
pip3 install virtualenv
virtualenv ~/venvs/koopmans
source ~/venvs/koopmans/bin/activate
```

Note that `koopmans` requires python v3.7 or later. If your computer's version of python3 corresponds to an earlier version, install python v3.7 or later, and then direct `virtualenv` to create the virtual environment using that specific installation of python via

```
virtualenv ~/venvs/koopmans -p /usr/bin/python3.x
```

Fetching the submodules

Now, ensure you have downloaded the various `git` submodules. To do so, run `make submodules`, or equivalently

```
git submodule init
git submodule update
```

Compiling Quantum ESPRESSO

Then you need to compile the copies of `Quantum ESPRESSO`. To do this, run

```
make espresso MPIF90=<mpif90>
```

where `<mpif90>` should be replaced by the name of your chosen MPI Fortran90 compiler e.g. `MPIF90=mpiifort`. The code should automatically detect and link the requisite libraries. (If this fails you may need to manually compile the two versions of `Quantum ESPRESSO` contained in the `quantum_espresso/` directory.)

Adding Quantum ESPRESSO to your path

To add all of the Quantum ESPRESSO binaries to your path, run

```
sudo make install
```

By default this will copy the Quantum ESPRESSO binaries to `/usr/local/bin`. This requires sudo privileges. If you do not have sudo privileges, you can either (a) install the codes in a different location by running `make install PREFIX=/path/to/bin/` (substitute `/path/to/bin/` with any directory of your choosing that is on your path) or (b) append `bin/` from the current directory to your path.

Installing the workflow manager

Finally, install the python workflow manager, either via `make workflow`, or

```
python3 -m pip install --upgrade pip
python3 -m pip install -e .
```

CHAPTER FIVE

THE INPUT FILE

koopmans takes a single JSON file as an input. Here is an example, taken from *Tutorial 1*:

```
{
  "workflow": {
    "functional": "ki",
    "method": "dscf",
    "init_orbitals": "kohn-sham",
    "from_scratch": true,
    "alpha_numsteps": 1,
    "pseudo_library": "sg15"
  },
  "atoms": {
    "cell_parameters": {
      "vectors": [[14.1738, 0.0, 0.0],
                  [0.0, 12.0, 0.0],
                  [0.0, 0.0, 12.66]],
      "units": "angstrom",
      "periodic": false
    },
    "atomic_positions": {
      "units": "angstrom",
      "positions": [
        ["O", 7.0869, 6.0, 5.89],
        ["O", 8.1738, 6.0, 6.55],
        ["O", 6.0, 6.0, 6.55]
      ]
    }
  },
  "calculator_parameters": {
    "ecutwfc": 65.0,
    "ecutrho": 260.0,
    "nbnd": 10
  }
}
```

As you can see, the file is divided into three blocks: *workflow*, *atoms*, and *calculator_parameters*. Other valid blocks are *kpoints*, *pseudopotentials*, and *plotting*. These are all explained below.

5.1 The workflow block

The `workflow` block contains variables that define the details of the workflow that we are going to run.

5.1.1 Valid keywords

5.2 The atoms block

The `atoms` block contains details about the atomic positions and the simulation cell. It contains two subdictionaries

`atomic_positions`
contains the atomic positions e.g.

```
"atomic_positions": {
  "units": "angstrom",
  "positions": [
    ["O", 7.0869, 6.0, 5.89],
    ["O", 8.1738, 6.0, 6.55],
    ["O", 6.0, 6.0, 6.55]
  ]
}
```

Valid options for `units` are `alat`, `angstrom`, `bohr`, and `crystal`

`cell_parameters`
describes the simulation cell. This can be specified explicitly

```
"cell_parameters": {
  "vectors": [[14.1738, 0.0, 0.0],
              [0.0, 12.0, 0.0],
              [0.0, 0.0, 12.66]],
  "units": "angstrom",
  "periodic": false
},
```

(with units `angstrom`, `bohr`, or `alat`), or using `ibrav` and `celldms` following the conventions of Quantum ESPRESSO e.g.

```
"cell_parameters": {
    "periodic": true,
    "ibrav": 2,
    "celldms": {"1": 10.2622}
},
```

5.3 The kpoints block

The `k_points` block specifies the k-point sampling e.g.

```
"kpoints": {
    "grid": [2, 2, 2],
    "offset": [0, 0, 0],
    "path": "LGXKG"
},
```

There are five possible entries in this block

grid
a list of three integers specifying the shape of the regular grid of k-points

offset
a list of three integers, either `0` or `1`. If `1`, the regular k-point grid is offset by half a grid step in that dimension

path
the path to be used in band structure plots, specified as a string of characters corresponding to special points of the Bravais lattice

density
the number k-points per inverse Angstrom along the path

gamma_only
set to `True` if the calculation is only sampling the gamma point

5.4 The pseudopotentials block

`koopmans` ships with several pre-installed pseudopotential libraries. To use these, select a `pseudo_library` in the `workflow` block. Valid options include `pseudo_dojo_standard/stringent` and `sg15`.

Alternatively, you can provide your own pseudopotentials via the `pseudopotentials` block, where we specify the filenames of the pseudopotentials for each element e.g.

```
"pseudopotentials": {"O": "O.upf", "H": "H.upf"}
```

The directory that these pseudopotentials are contained in should be provided via the `pseudo_directory` keyword in the `workflow` block. See *here for more details on setting up pseudopotentials*.

Warning: Koopmans currently only works with norm-conserving pseudopotentials

5.5 The calculator_parameters block

The `calculator_parameters` block can be used to specify code-specific codes e.g.

```
"calculator_parameters": {
    "ecutwfc": 60.0,
    "pw": {
        "system": {
```

(continues on next page)

(continued from previous page)

```
            "nbnd": 20
        }
    },
    "w90": {
        "bands_plot": true,
        "projections": [[{"fsite": [ 0.25, 0.25, 0.25 ], "ang_mtm": "sp3"}],
                        [{"fsite": [ 0.25, 0.25, 0.25 ], "ang_mtm": "sp3"}]],
        "dis_froz_max": 10.6,
        "dis_win_max": 16.9
    },
    "ui": {
        "smooth_int_factor": 4
    }
},
```

Note that any keyword specified outside of a subblock (e.g. `ecutwfc` in the above example) is applied to all calculators for which it is recognized keyword.

Note: Generally, you will need to provide very few keywords in this block. The vast majority of keywords for each calculation will be generated automatically. Notable exceptions to this are `ecutwfc` and (if relevant) the `Wannier90` projection

5.5.1 The pw subblock

This subblock contains keywords specific to `pw.x` (see the list of valid pw.x keywords). Note that they should not be grouped by namelists as they are in a `pw.x` input file.

5.5.2 The w90 subblock

This subblock contains keywords specific to `wannier90.x`, which are documented here. The one keyword for which the syntax differs is the `projections` block, via which the user specifies the projections used to initialize the Wannier functions.

An individual projection can be specified as either a dictionary or a string.

As a dictionary

If specifying a projection via a dictionary, the required entries for this dictionary are

`site/csite/fsite`

an atom label/cartesian coordinate/fractional coordinate to be used as the projections' center. The three are mutually exclusive.

`ang_mtm`

a string specifying the angular momentum states e.g. `"l=2"`

The user can also optionally specify `zaxis`, `xaxis`, `radial`, `zona` (see the Wannier90 User Guide for details).

As a string

If specifying a projection via a string, this string must follow the `Wannier90` syntax e.g. `"f=0.25,0.25,0.25:sp3"`

These individual projections (either as dictionaries or as strings) must be provided to `projections` within a list of lists. This is because for Koopmans calculations, we want to perform the Wannierization in quite a particular way

- the occupied and empty manifolds must be wannierized separately.
- the occupied or empty manifold can consist of several well-separated blocks of bands. In this instance it is desirable to Wannierize each block separately, preventing the Wannierization procedure from mixing bands that are far apart in energy space.

We can achieve both of the above via the list-of-lists syntax. Consider the following example for the wannierization of bulk ZnO

```
"w90": {
    "projections": [
        [{"site": "Zn", "ang_mtm": "l=0"}],
        [{"site": "Zn", "ang_mtm": "l=1"}],
        [{"site": "O", "ang_mtm": "l=0"}],
        [{"site": "Zn", "ang_mtm": "l=2"},
         {"site": "O", "ang_mtm": "l=1"}],
        [{"site": "Zn", "ang_mtm": "l=0"}]
    ],
```

In ZnO, the bands form several distinct blocks. The first block of occupied bands have Zn 3s character, the next Zn 3p, then O 2s, and finally Zn 3d hybridized with O 2p. The first empty bands have Zn 4s character. You can see this reflected in the way the projections have been specified. If we were to run the workflow with this configuration, it will run five separate Wannierizations, one for each sub-list.

This means that

- the occupied and empty manifolds will be wannierized separately, because the cumulative number of projections in the first four blocks is commensurate with the number of occupied bands in our system
- we prevent mixing bands that belong to different sub-lists

See *here for a more detailed tutorial on projections*.

Note: The order of the projections blocks is important: they must run from lowest-energy to highest-energy.

Note: If disentanglement keywords such as `dis_win_max` are provided, these will only be used during the Wannierization of the final block of projections

5.5.3 The pw2wannier subblock

This subblock contains `pw2wannier90.x` keywords, in a single dictionary with no subdictionaries.

5.5.4 The kcp subblock

This subblock contains keywords specific to `kcp.x`, a modified version of `cp.x` for performing Koopmans calculations. In addition to the keywords associated with cp.x there are several new keywords associated with the Koopmans implementation in `kcp.x`. Non-experts will never need to change these.

5.5.5 The ui subblock

This subblock controls the unfolding and interpolation procedure for generating band structures and densities of states from -only supercell calculations.

Valid keywords

5.6 The convergence block

This block can be used to customize a convergence calculation.

5.6.1 Valid keywords

See *here for a more detailed tutorial on performing convergence calculations*.

5.7 The plotting block

This block can be used to customize the band structures and densities of states plots that `koopmans` generates.

5.7.1 Valid keywords

5.8 The ml block

Warning: This feature is experimental

This block controls the machine-learning of screening parameters.

5.8.1 Valid keywords

CHAPTER

SIX

HOW TO RUN

To run a calculation from the command line, all that is required is

```
$ koopmans <seed>.json
```

where `<seed>.json` is a `koopmans` input file. The format of the input file is documented *here*.

6.1 Running via python

It is possible to run `koopmans` workflows from within python, bypassing the need for an input file entirely. To do this, all you need to do is create a `SinglepointWorkflow` object

```
wf = SinglepointWorkflow(...)
```

and then simply call

```
wf.run()
```

For details of how to initialize a workflow object, see the *workflow class documentation*. After a calculation has finished, you can access the individual calculations e.g.

```
final_calc = wf.calculations[-1]
```

and fetch their results e.g.

```
total_energy = final_calc.results['energy']
```

6.2 Parallelism

In order to run the code in parallel, define the environment variables `PARA_PREFIX` and `PARA_POSTFIX`. These are defined in the same way as in `Quantum ESPRESSO`, e.g.

```
export PARA_PREFIX="srun"
export PARA_POSTFIX="-npool 4"
```

6.3 Pseudopotentials

Currently, Koopmans functionals only works with norm-conserving pseudopotentials. We suggest you use optimized norm-conserving Vanderbilt pseudopotentials, such as

- the SG15 library
- the Pseudo Dojo library

For convenience, `koopmans` already ships with both of these pseudopotential libraries and you can simply select the one you want to use using the `pseudo_library` keyword.

If you prefer to use your own pseudopotentials, add them to `src/koopmans/pseudopotentials/<my_pseudos>/<functional>`, where `<my_pseudos>` is a name of your choosing and `<functional>` is the functional used to generate your pseudopotentials. You can then direct `koopmans` to use these pseudopotentials by setting the keywords `pseudo_library` and `base_functional` to `<my_pseudos>` and `<functional>` respectively.

Alternatively, you can direct the code to always use your personal pseudopotentials directory by defining the variable

```
export ESPRESSO_PSEUDO="/path/to/pseudopotential/folder/"
```

CHAPTER

SEVEN

MODULES

In this section we document the various classes defined within the `koopmans` python package.

7.1 The workflow module

The central objects in `koopmans` are `Workflow` objects. We use these to define and run workflows, as described *here*. The workflow that runs most calculations is the `SinglepointWorkflow`, defined as follows.

class `koopmans.workflows.`**`SinglepointWorkflow`**(*atoms*, *pseudopotentials={}*, *kpoints=None*, *projections=None*, *name='koopmans_workflow'*, *parameters={}*, *calculator_parameters=None*, *plotting={}*, *ml={}*, *autogenerate_settings=True*, *version=None*, ***kwargs*)

Abstract base class that defines a Koopmans workflow

Parameters

atoms
[Atoms] an ASE `Atoms` object defining the atomic positions, cell, etc

pseudopotentials
[Dict[str, str]] a dictionary mapping atom labels to pseudopotential filenames

kpoints
[koopmans.kpoints.Kpoints] a dataclass defining the k-point sampling and paths

projections
[ProjectionsBlocks] The projections to be used in the Wannierization

name
[str] a name for the workflow

parameters
[Dict[str, Any] | koopmans.settings.WorkflowSettingsDict] a dictionary specifying any workflow settings to use; note that a simpler alternative is to provide workflow settings as keyword arguments

calculator_parameters
[Dict[str, koopmans.settings.SettingsDict]] a dictionary containing calculator-specific settings; as for the parameters, it is usually simpler to specify these individually as keyword arguments

plotting
[koopmans.settings.PlotSettingsDict] a dictionary containing settings specific to plotting; again, it is usually simpler to specify these individually as keyword arguments

autogenerate_settings
[bool] if True (the default), autogenerate various calculator settings; the only scenario where you do not want to do this is when creating a new workflow from a .kwf file

****kwargs**
any valid workflow, calculator, or plotting settings e.g. `{"functional": "ki", "ecutwfc": 50.0}`

Examples

Running a Koopmans calculation on ozone

```
>>> from ase.build import molecule
>>> from koopmans.workflows import SinglepointWorkflow
>>> ozone = molecule('O3', vacuum=5.0, pbc=False)
>>> wf = SinglepointWorkflow(ozone, ecutwfc = 20.0)
>>> wf.run()
```

Running a Koopmans calculation on GaAs

```
>>> from ase.build import bulk
>>> from koopmans.projections import ProjectionBlocks
>>> from koopmans.kpoints import Kpoints
>>> from koopmans.workflows import SinglepointWorkflow
>>> gaas = bulk('GaAs', crystalstructure='zincblende', a=5.6536)
>>> projs = ProjectionBlocks.fromlist([["Ga: d"], ["As: sp3"], ["Ga: sp3"]],
>>>                                   spins=[None, None, None],
>>>                                   atoms=gaas)
>>> kpoints = Kpoints(grid=[2, 2, 2])
>>> wf = SinglepointWorkflow(gaas, kpoints=kpoints, projections=projs, init_
↪orbitals='mlwfs',
>>>                          pseudo_library='sg15_v1.0', ecutwfc=40.0,
>>>                          calculator_parameters={'pw': {'nbnd': 45},
>>>                          'w90_emp': {'dis_froz_max': 14.6, 'dis_win_max': 18.6}}
↪)
>>> wf.run()
```

CHAPTER EIGHT

TUTORIALS

8.1 Tutorial 1: the ionization potential and electron affinity of ozone

In this tutorial, we will calculate the ionisation potential and electron affinity of ozone.

8.1.1 The input

The input file for this calculation can be downloaded `here`. Just to briefly highlight the most important details of the workflow block

```
  "workflow": {
    "functional": "ki",
    "method": "dscf",
```

here we select the KI functional (as opposed to KIPZ),

```
    "functional": "ki",
    "method": "dscf",
    "init_orbitals": "kohn-sham",
```

specifies that we are going to calculate the screening parameters via a SCF procedure, whereby we compute the energies of various N, $N-1$, and $N+1$-electron systems (see *the theory section* for details), and

```
    "method": "dscf",
    "init_orbitals": "kohn-sham",
    "from_scratch": true,
```

specifies that we have chosen to use the Kohn-Sham orbitals as our *variational orbitals*. This is common practice for molecules.

Meanwhile, the `atoms` block describes the both the cell and the atoms it contains. If you are familiar with `Quantum ESPRESSO` input files then most of this should look very familiar to you (albeit in JSON format).

8.1.2 Running the calculation

In order to run the calculation, simply run

```
koopmans ozone.json | tee ozone.out
```

Tip: In order to run in parallel, set the `PARA_PREFIX` environment variable to `mpirun -np 4` (or similar)

8.1.3 The output

First, let us inspect the contents of `ozone.out`: after the header we can see there are a list of Quantum ESPRESSO calculations that have been run by `koopmans`. These come under three main headings.

Initialization

The first step in any Koopmans calculation is the initialization step. In this step we initialize the density and the variational orbitals.

```
Initialisation of density and variational orbitals
==================================================
 Running init/dft_init_nspin1... done
 Running init/dft_init_nspin2_dummy... done
 Running init/dft_init_nspin2... done
 Overwriting the variational orbitals with Kohn-Sham orbitals
 Copying the spin-up variational orbitals over to the spin-down channel
```

For this calculation we can see that `koopmans` has run three PBE calculations. These initialize the density with the PBE density. Indeed, from this point onwards in the calcuation the density will never change, because the KI functional yields the same density as the base functional. (N.B. This is not true of KIPZ.)

These PBE calculations also have provided us with our variational orbitals – we can see that the calculation has selected the PBE Kohn-Sham orbitals as the variational orbitals (because earlier we set `"init_orbitals": "kohn-sham"`).

But why three PBE calculations? The reason for this is that the calculations we will perform later involve the addition/removal of a single electron, which means the density we need to generate here must correspond to a `nspin = 2` calculation. However, we know ozone is a closed-shell molecule and simply performing a `nspin = 2` PBE calculation risks introducing spin contamination (i.e. falling into a local minimum where $n^\uparrow(\mathbf{r}) \neq n^\downarrow(\mathbf{r})$).

This sequence of three calculations is designed to avoid this; we first optimise the density constrained to be spin-unpolarized, and only once that density has been minimised do we lift this restriction. This additional step can be disabled by adding `"fix_spin_contamination": false` to the `workflow` block of `ozone.json`.

The input and output Quantum ESPRESSO files for this first step can all be found in the directory `init/`.

Calculating the screening parameters

The second step in the calculation involves the calculation of the screening parameters:

```
Calculating screening parameters
================================
 Running calc_alpha/ki... done

Orbital 1
---------
 Running calc_alpha/orbital_1/dft_n-1... done

Orbital 2
---------
 Running calc_alpha/orbital_2/dft_n-1... done

Orbital 3
---------
 Running calc_alpha/orbital_3/dft_n-1... done

Orbital 4
---------
 Running calc_alpha/orbital_4/dft_n-1... done
```

etc. Here, we are calculating the screening parameters using the *SCF method*. For filled orbitals (orbitals 1-9 of ozone) this requires an $N-1$-electron PBE calculation where we freeze the i^{th} orbital, empty it, and allow the rest of the density to relax. This yields $E_i(N-1)$. Meanwhile, $E(N)$, $\varepsilon_i^\alpha(1)$, and $\varepsilon_i^0(1)$ are all obtained during the trial KI calculation `calc_alpha/ki`. Together with the value of our guess for the screening parameters ($\alpha_i^0 = 0.6$), this is sufficient to update our guess for α_i (see the *theory section* for details).

The procedure for empty orbitals is slightly different, as we can see when it comes to orbital 10:

```
Orbital 10
----------
 Running calc_alpha/orbital_10/pz_print... done
 Running calc_alpha/orbital_10/dft_n+1_dummy... done
 Running calc_alpha/orbital_10/dft_n+1... done

```

where now we must call Quantum ESPRESSO several times in order to obtain $E_i(N+1)$.

pz_print and dft_n+1_dummy
    preliminary calculations that generate files required by the subsequent constrained DFT calculation

dft_n+1
    a $N+1$-electron PBE calculation where we freeze the 10th orbital, fill it, and allow the rest of the density to relax. This yields $E_i(N+1)$

At the end of this section we can see a couple of tables:

```

 alpha
        1        2        3        4   ...       7        8        9       10
 0  0.60000  0.600000  0.600000  0.600000  ...  0.600000  0.600000  0.600000  0.60000
 1  0.65568  0.727565  0.783855  0.663858  ...  0.729884  0.741895  0.779259  0.71739

```

(continues on next page)

(continued from previous page)

```
   [2 rows x 10 columns]

   Delta E_i - epsilon_i (eV)
            1         2         3   ...        8        9        10
   0 -0.477272 -0.920449 -1.124074  ... -0.888568 -1.05013  0.711214

   [1 rows x 10 columns]

   Screening parameters have been determined but are not necessarily converged
```

The first table lists the screening parameters α_i that we obtained – we can see from row 0 we started with a guess of $\alpha_i = 0.6$ for every orbital i, and row 1 shows the alpha values.

The second table lists $\Delta E_i - \varepsilon_i^\alpha$. This is a measure of how well converged the alpha values are: if this value is close to zero, then the alpha values are well-converged. Note that the values listed above correspond to our starting guess of $\alpha_i = 0.6$; this table does not show how well-converged the final alpha values are.

Note: In order to see how well-converged our new screening parameters are, try increasing `alpha_numsteps` in the input file from 1 to 2. Can you make sense of the contents of the resulting tables?

The input and output Quantum ESPRESSO files for this step can be found in the directory `calc_alpha/`.

The final calculation

Having determined the screening parameters, the final KI calculation is now run:

```
  Final KI calculation
  ====================
   Running final/ki_final... done

 Workflow complete
```

The input and output Quantum ESPRESSO files for this step can be found in the directory `final/`.

8.1.4 Extracting the ionisation potential and electron affinity

Let's now extract the KI ionisation potential and electron affinity for ozone from our calculation.

The ionisation potential (IP) corresponds to the negative of the energy of the HOMO (highest occupied molecular orbital). If you open `final/ki_final.cpo` and search near the bottom of the file you will see a section something like

```
...
HOMO Eigenvalue (eV)

-12.5199

LUMO Eigenvalue (eV)

-1.8218

```

(continues on next page)

(continued from previous page)

```
Eigenvalues (eV), kp =   1 , spin =  1

-40.1869  -32.9130  -24.2288  -19.6841  -19.4902  -19.2696  -13.6037  -12.7618  -12.5199

Empty States Eigenvalues (eV), kp =   1 , spin =  1

-1.8218

Electronic Gap (eV) =    10.6981


Eigenvalues (eV), kp =   1 , spin =  2

-40.1869  -32.9130  -24.2288  -19.6841  -19.4902  -19.2696  -13.6037  -12.7618  -12.5199

Empty States Eigenvalues (eV), kp =   1 , spin =  2

-1.8218

Electronic Gap (eV) =    10.6981
...
```

Very clearly we can see the HOMO eigenvalue of -12.52 eV. Thus we have a KI IP of 12.52 eV. This compares extremely well to the experimental value of ~ 12.5 eV, and is a marked improvement on the PBE result of 7.95 eV (which we can obtain from the `HOMO Eigenvalue` in `init/dft_init_nspin2.cpo`).

Meanwhile, the electron affinity (EA) corresponds to the negative of the energy of the LUMO (lowest unoccupied molecular orbital). From the same section in `final/ki_final.cpo` we can see that the KI EA is 1.82 eV (cf. ~ 2.1 eV experiment, 6.17 eV PBE)

Tip: If you prefer working within `python`, you need not write a script to parse the contents of `final/ki_final.cpo` in order to extract the IP and EA. Instead, `koopmans` will have generated a python-readable file `ozone.kwf` containing all of the important calculation data.

You can read these files like so:

```python
from koopmans import io

# Load the workflow object
wf = io.read('ozone.kwf')

# Access the results from the very last calculation
results = wf.calculations[-1].results

# Calculate the IP and EA
ip = -results['homo_energy']
ea = -results['lumo_energy']

# Print
print(f' IP = {ip:.2f} eV')
print(f' EA = {ea:.2f} eV')
```

Indeed, it is also possible to run the workflow from within `python` (rather than calling `koopmans` from the command line)

```
from koopmans.io import read

wf = read('ozone.json')
wf.run()
```

in which case you have immediate access to the workflow object `wf` rather than having to load in the `.kwf` file.

8.2 Tutorial 2: the band structure of bulk silicon (calculated via a supercell)

In this tutorial, we will calculate the KI bandstructure of bulk silicon. The input file used for this calculation can be downloaded `here`.

8.2.1 Wannierization

While this calculation will bear a lot of similarity to the previous tutorial, there are several differences between performing a Koopmans calculation on a bulk system vs. a molecule. One major difference is that we use Wannier functions as our variational orbitals.

What are Wannier functions?

Most electronic-structure codes try to calculate the Bloch states $\psi_{n\mathbf{k}}$ of periodic systems (where n is the band index and $\mathbf{k}$ the crystal momentum). However, other representations are equally valid. One such representation is the Wannier function (WF) basis. In contrast to the very delocalised Bloch states, WFs are spatially localised and as such represent a very convenient basis to work with. In our case, the fact that they are localised means that they are suitable for use as variational orbitals.

Wannier functions $w_{n\mathbf{R}}(\mathbf{r})$ can be written in terms of a transformation of the Bloch states:

$$w_{n\mathbf{R}}(\mathbf{r}) = \frac{V}{(2\pi)^3}\int_{\mathrm{BZ}}\left[\sum_m U^{(\mathbf{k})}_{mn}\psi_{m\mathbf{k}}(\mathbf{r})\right]e^{-i\mathbf{k}\cdot\mathbf{R}}\mathrm{d}\mathbf{k} \quad (8.1)$$

where our Wannier functions belong to a particular lattice site $\mathbf{R}$, V is the unit cell volume, the integral is over the Brillouin zone (BZ), and $U^{(\mathbf{k})}_{mn}$ defines a unitary rotation that mixes the Bloch states with crystal momentum $\mathbf{k}$. Crucially, this matrix $U^{(\mathbf{k})}_{mn}$ is not uniquely defined — indeed, it represents a freedom of the transformation that we can exploit.

We choose our $U^{(\mathbf{k})}_{mn}$ that gives rise to WFs that are “maximially localised”. We quantify the spread Ω of a WF as

$$\Omega = \sum_n\left[\langle w_{n\mathbf{0}}(\mathbf{r})|\, r^2\, |w_{n\mathbf{0}}(\mathbf{r})\rangle - |\langle w_{n\mathbf{0}}(\mathbf{r})|\, \mathbf{r}\, |w_{n\mathbf{0}}(\mathbf{r})\rangle|^2\right] \quad (8.2)$$

The Wannier functions that minimise this spread are called maximally localised Wannier functions (MLWFs). For more details, see Ref. [20].

How do I calculate Wannier functions?

MLWFs can be calculated with Wannier90, an open-source code that is distributed with `Quantum ESPRESSO`. Performing a Wannierization with Wannier90 requires a series of calculations to be performed with `pw.x`, `wannier90.x`, and `pw2wannier90.x`. This workflow is automated within `koopmans`, as we will see in this tutorial.

Note: This tutorial will not discuss in detail how perform a Wannierization with Wannier90. The Wannier90 website contains lots of excellent tutorials.

One important distinction to make for Koopmans calculations — as opposed to many of the Wannier90 tutorials — is that we need to separately Wannierize the occupied and empty manifolds.

Let's now inspect this tutorial's `input file`. At the top you will see that

```
    "workflow": {
        "task": "wannierize",
        "functional": "ki",
```

which tells the code to perform a standalone Wannierization calculation. Meanwhile, near the bottom of the file there are some Wannier90-specific parameters provided in the `w90` block

```
        "w90": {
            "bands_plot": true,
            "projections": [[{"fsite": [ 0.25, 0.25, 0.25 ], "ang_mtm": "sp3"}],
                            [{"fsite": [ 0.25, 0.25, 0.25 ], "ang_mtm": "sp3"}]],
            "dis_froz_max": 10.6,
            "dis_win_max": 16.9
        },
```

The `w90` block format is explained more fully *here*.

We run this calculation as per usual:

```
koopmans si.json | tee si_wannierize.out
```

After the usual header, you should see something like the following:

```

 Wannierization
 ==============
  Running wannier/scf... done
  Running wannier/nscf... done
  Running wannier/block_1/wann_preproc... done
  Running wannier/block_1/pw2wan... done
  Running wannier/block_1/wann... done
  Running wannier/block_2/wann_preproc... done
  Running wannier/block_2/pw2wan... done
  Running wannier/block_2/wann... done
  Running wannier/bands... done
  Running pdos/projwfc... done
```

(continues on next page)

(continued from previous page)

```

Workflow complete
```

These various calculations that are required to obtain the MLWFs of bulk silicon. You can inspect the and various output files will have been generated in a new `wannier/` directory.

scf
: a `pw.x` self-consistent DFT calculation performed with no empty bands. This obtains the ground-state electronic density

nscf
: a `pw.x` non-self-consistent DFT calculation that determines the Hamiltonian, now including some empty bands

block_1/wann_preproc
: a preprocessing `wannier90.x` calculation that generates some files required by `pw2wannier90.x`

block_1/pw2wan
: a `pw2wannier90.x` calculation that extracts from the eariler `pw.x` calculations several key quantities required for generating the Wannier orbitals for the occupied manifold: namely, the overlap matrix of the cell-periodic part of the Block states (this is the `wann.mmn` file) and the projection of the Bloch states onto some trial localised orbitals (`wann.amn`)

block_1/wann
: the `wannier90.x` calculation that obtains the MLWFs for the occupied manifold

block_2/...
: the analogous calculations as those in `occ/`, but for the empty manifold

bands
: a `pw.x` calculation that calculates the band structure of silicon explicitly, used for verification of the Wannierization (see the next section)

pdos/projwfc
: a `projwfc.x` calculation that calculates the projected density of states, also used for checking the Wannierization

The main output files of interest in `wannier/` are files `block_1/wann.wout` and `block_2/wann.wout`, which contain the output of `wannier90.x` for the Wannierization of the occupied and empty manifolds. If you inspect either of these files you will be able to see a lot of starting information, and then under a heading like

```
*------------------------------- WANNIERISE ---------------------------------*
+--------------------------------------------------------------------+<-- CONV
| Iter  Delta Spread     RMS Gradient      Spread (Ang^2)      Time  |<-- CONV
+--------------------------------------------------------------------+<-- CONV
```

you will then see a series of steps where you can see the Wannier functions being optimised and the spread (labelled `SPRD`) decreasing from one step to the next. Scrolling down further you should see a statement that the Wannierization procedure has converged, alongside with a summary of the final state of the WFs.

How do I know if the Wannier functions I have calculated are "good"?

Performing a Wannierization calculation is not a straightforward procedure, and requires the tweaking of the Wannier90 input parameters in order to obtain a "good" set of Wannier functions.

One check is to see if an interpolated bandstructure generated by the MLWFs resembles an explicitly-calculated band structure. (For an explanation of how one can use Wannier functions to interpolate band structures, refer to Ref. [20].) You might have noticed that when we ran `koopmans` earlier it also generated a file called `si_wannierize_bandstructure.png`. It should look something like this:

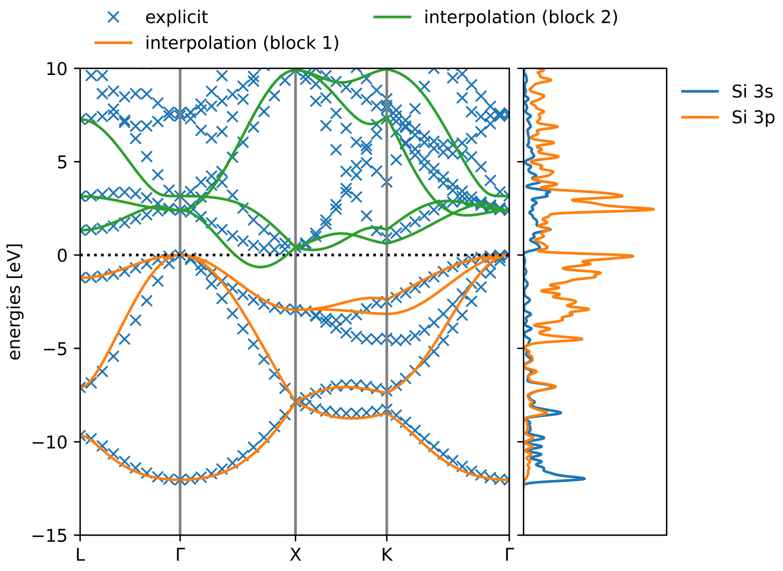


Fig. 1: Comparing the interpolated and explicitly-calculated band structures of bulk silicon

Clearly, we can see that the interpolation is no good! (The interpolated band structure ought to lie on top of the explicitly calculated band structure.) The reason for this is that the Brillouin zone is undersampled by our $2 \times 2 \times 2$ k-point grid. Try increasing the size of the k-point grid and see if the interpolated bandstructure improves.

Tip: Trying different grid sizes can be very easily automated within `python`. Here is a simple script that will run the Wannierization for three different grid sizes:

```
from koopmans.io import read
from koopmans.utils import chdir

for grid_size in [2, 4, 8]:

    # Read in the input file
    wf = read('si.json')

    # Modify the kgrid
    wf.kpoints.grid = [grid_size, grid_size, grid_size]

    # Run the workflow in a subdirectory
    with chdir('{0}x{0}x{0}'.format(grid_size)):
        wf.run()
```

Tip: You may also notice a file called `si_wannierize_bandstructure.fig.pkl`. This is a version of the figure that you can load in python and modify as you see fit. e.g. here is a script that changes the y-axis limits and label:

```
import pickle

import matplotlib.pyplot as plt

with open('2x2x2/si_wannierize_bandstructure.fig.pkl', 'rb') as fd:
    pickle.load(fd)

ax = plt.gca()
ax.set_ylim([-5, 5])
ax.set_ylabel(r'$\omega$ (eV)')
# plt.show() uncomment to view the figure interactively
plt.savefig('2x2x2/si_wannierize_bandstructure_rescaled.png')
```

8.2.2 The KI calculation

Having obtained a Wannierization of silicon that we are happy with, we can proceed with the KI calculation. In order to do this simply change the `task` keyword in the input file from `wannierize` to `singlepoint`.

Tip: Although we just discovered that a $2 \times 2 \times 2$ k-point grid was inadequate for producing good Wannier functions, this next calculation is a lot more computationally intensive and will take a long time on most desktop computers. We therefore suggest that for the purposes of this tutorial you switch back to the small k-point grid. (But for any proper calculations, always use high-quality Wannier functions!)

Initialization

If you run this new input the output will be remarkably similar to that from the previous tutorial, with a couple of exceptions. At the start of the workflow you will see there is a Wannierization procedure, much like we had earlier when we running with the `wannierize` task:

```
   Wannierization
   ==============
    Running wannier/scf... done
    Running wannier/nscf... done
    Running wannier/block_1/wann_preproc... done
    Running wannier/block_1/pw2wan... done
    Running wannier/block_1/wann... done
    Running wannier/block_2/wann_preproc... done
    Running wannier/block_2/pw2wan... done
    Running wannier/block_2/wann... done
```

which replaces the previous series of semi-local and PZ calculations that we used to initialize the variational orbitals for a molecule.

There is then an new “folding to supercell” subsection:

```
  Folding to supercell
  --------------------
   Running block_1/w2kcp... done
   Running block_2/w2kcp... done
```

In order to understand what these calculations are doing, we must think ahead to the next step in our calculation, where we will calculate the screening parameters using the SCF method. These calculations, where we remove/add an electron from/to the system, require us to work in a supercell. This means that we must transform the k-dependent primitive cell results from previous calculations into equivalent Γ-only supercell quantities that can be read by `kcp`. This is precisely what the above `wan2odd` calculations do.

Calculating the screening parameters

Having transformed into a supercell, the calculation of the screening parameters proceeds as usual. The one difference to tutorial 1 that you might notice at this step is that we are skipping the calculation of screening parameters for some of the orbitals:

```
Calculating screening parameters
================================
 Running calc_alpha/ki... done

Orbitals 1-31
-------------
 Skipping; will use the screening parameter of an equivalent orbital

Orbital 32
----------
```

The code is doing this because of what we provided for the `orbital_groups` in the input file:

```
      "alpha_guess": 0.077,
      "orbital_groups": [0, 0, 0, 0, 1, 1, 1, 1],
      "pseudo_library": "pseudo_dojo_standard",
```

which tells the code to use the same parameter for orbitals belonging to the same group. In this instance we are calculating a single screening parameter for all four filled orbitals, and a single screening parameter for the empty orbitals.

The final calculation and postprocessing

The final difference for the solids calculation is that there is an additional preprocessing step at the very end:

```
Postprocessing
===============


  Wannierization
  ==============
   Running wannier/scf... done
   Running wannier/nscf... done
   Running wannier/block_1/wann_preproc... done
   Running wannier/block_1/pw2wan... done
   Running wannier/block_1/wann... done
```

(continues on next page)

(continued from previous page)

```
     Running wannier/block_2/wann_preproc... done
     Running wannier/block_2/pw2wan... done
     Running wannier/block_2/wann... done
     Running wannier/bands... done
     Running pdos/projwfc... done
    Running occ/ki... done
    Running emp/ki... done

Workflow complete
```

Here, we transform back our results from the supercell sampled at Γ to the primitive cell with k-space sampling. This allows us to obtain a bandstructure. The extra Wannierization step that is being performed is to assist the interpolation of the band structure in the primitive cell, and has been performed because in the input file we specified

```
        "ui": {
            "smooth_int_factor": 4
        }
```

For more details on the "unfold and interpolate" procedure see *here* and Ref. [11].

8.2.3 Extracting the KI bandstructure and the bandgap of Si

The bandstructure can be found in `postproc/bands_interpolated.dat` as a raw data file, but there is a more flexible way for plotting the final bandstructure using the python machinery of `koopmans`:

```python
from koopmans.io import read

# Load the workflow object
wf = read('si.kwf')

# Access the band structure from the very last calculation
results = wf.calculations[-1].results
bs = results['band structure']

# Print the band strucutre to file
bs.plot(filename='si_bandstructure.png')

# Extract the band gap
n_occ = wf.number_of_electrons() // 2
gap = bs.energies[:, :, n_occ:].min() - bs.energies[:, :, :n_occ].max()
print(f'KI band gap = {gap:.2f} eV')
```

Running this script will generate a plot of the bandstructure (`si_bandstructure.png`) as well as printing out the band gap. You should get a result around 1.35 eV. Compare this to the PBE result of 0.68 eV and the experimental value of 1.22 eV. If we were more careful with the Wannier function generation, our result would be even closer (indeed in Ref. [24] the KI band gap was found to be 1.22 eV!)

8.3 Tutorial 3: the band structure of ZnO (calculated with explicit k-points)

In this tutorial we will calculate the band structure of bulk zinc oxide using the k-space formulation of Koopmans. The input file for this tutorial can be downloaded here.

8.3.1 Calculating the Koopmans band structure

The input file

First, let us inspect the input file:

```
{
    "workflow": {
        "task": "singlepoint",
        "functional": "ki",
        "base_functional": "lda",
        "method": "dfpt",
        "init_orbitals": "mlwfs",
        "calculate_alpha" : false,
        "alpha_guess": [[0.3580, 0.3641, 0.3640, 0.3641, 0.3571, 0.3577, 0.3577, 0.3573,␣
↪0.3573, 0.3580, 0.3641, 0.3640, 0.3641, 0.3571, 0.3577, 0.3577, 0.3573, 0.3573, 0.2158,
↪ 0.2323, 0.2344, 0.2343, 0.2158, 0.2323, 0.2344, 0.2343, 0.2231, 0.2231]],
```

Here we tell the code to calculate the KI bandstructure using the DFPT primitive cell approach. We will not actually calculate the screening parameters in this tutorial (because this calculation takes a bit of time) so we have set `calculate_alpha` to `False` and we have provided some pre-computed screening parameters in the `alpha_guess` field.

The rest of the file contains the atomic coordinates, k-point configuration, and calculator parameters (including the Wannier projectors, which we will discuss later).

Running the calculation

Running `koopmans zno.json` should produce an output with several sections: after the header there is the Wannierization

```
  Wannierization
  ==============
   Running wannier/scf... done
   Running wannier/nscf... done
   Running wannier/block_1/wann_preproc... done
   Running wannier/block_1/pw2wan... done
   Running wannier/block_1/wann... done
   Running wannier/block_2/wann_preproc... done
   Running wannier/block_2/pw2wan... done
   Running wannier/block_2/wann... done
   Running wannier/block_3/wann_preproc... done
   Running wannier/block_3/pw2wan... done
   Running wannier/block_3/wann... done
   Running wannier/block_4/wann_preproc... done
```

(continues on next page)

(continued from previous page)

```
    Running wannier/block_4/pw2wan... done
    Running wannier/block_4/wann... done
    Running wannier/block_5/wann_preproc... done
    Running wannier/block_5/pw2wan... done
    Running wannier/block_5/wann... done
    Running wannier/bands... done
    Running pdos/projwfc... done
```

which is very similar to what we saw for silicon, except now we have several blocks for the occupied manifold (discussed below). Then we have

```
  Conversion to Koopmans format
  -----------------------------
   Running wannier/kc... done
```

where the `Wannier90` files are converted into a format readable by the `kcw.x` code.

If we had instructed the code to calculate the alpha parameters, this would be followed by an extra block where these are calculated. But since we have already provided these, the workflow progresses immediately to the final step

```
  Construction of the Hamiltonian
  ===============================
   Running hamiltonian/kc... done

 Workflow complete
```

where the Koopmans Hamiltonian is constructed, and the band structure computed.

Plotting the results

Running the workflow will have produced several directories containing various input and output `Quantum ESPRESSO` files. It will also have generated `png` band structure plots, including this one of the Koopmans band structure:

However, suppose we want to make a nicer, more comprehensive plot comparing the LDA and Koopmans band structures. To achieve this, we will load all of the information from the `zno.kwf` file. This will provide us with a `SinglepointWorkflow` object which contains all of the calculations and their associated results. For example, we can access the calculations corresponding to the Koopmans and LDA band structures as follows:

```
from koopmans.io import read

# Load the workflow
wf = read('zno.kwf')

# The Koopmans bands were generated by the very last calculation in the workflow
koopmans_calc = wf.calculations[-1]

# The LDA bands were generated by a pw.x calculation with the setting "calculation =␣
↪bands"
[lda_calc] = [c for c in wf.calculations if c.parameters.get('calculation', None) ==
↪'bands']
```

and then the band structures are in the `results` dictionary of these calculators:

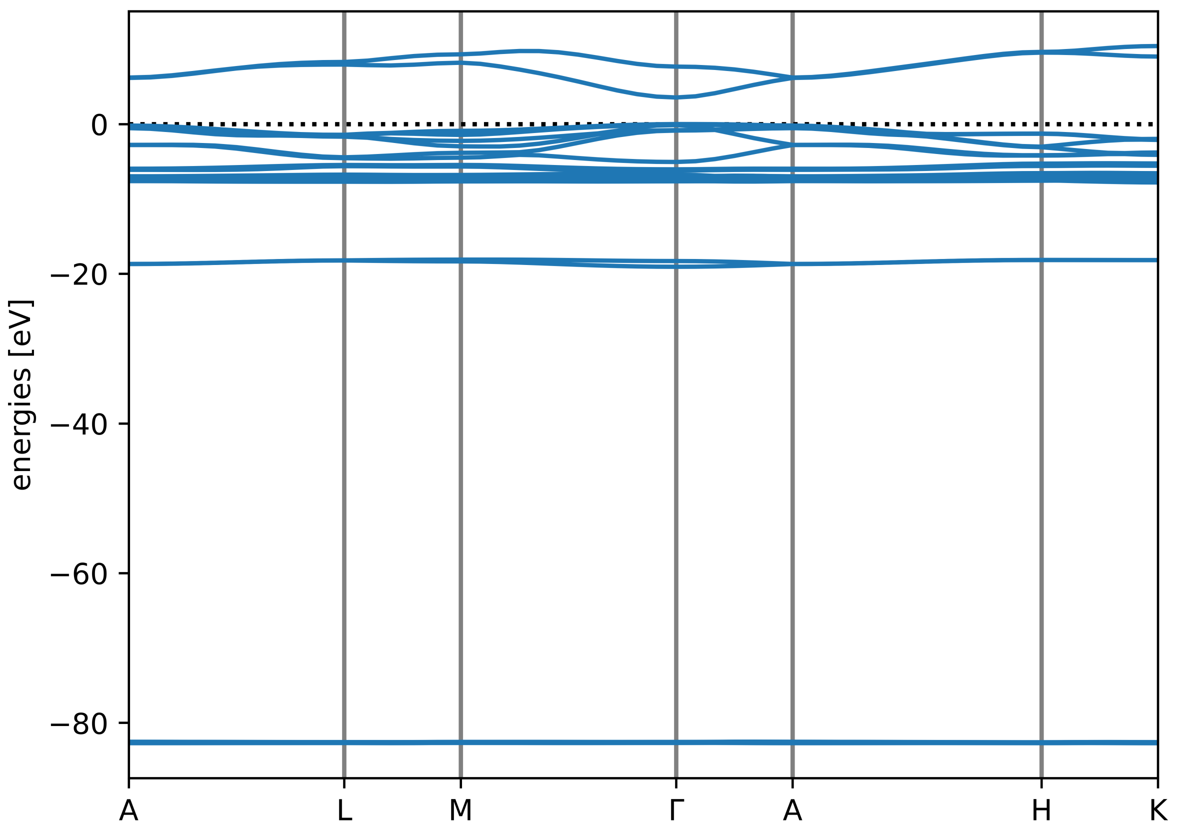


Fig. 2: The autogenerated band structure plot for ZnO

```
# Fetch the Koopmans bands, and shift them so that the valence band maximum is zero
koopmans_bs = koopmans_calc.results['band structure']
koopmans_bs_shifted = koopmans_bs.subtract_reference()

# Fetch the LDA bands, and shift them by the same amount
lda_bs = lda_calc.results['band structure']
lda_bs_shifted = lda_bs.subtract_reference(koopmans_bs.reference)
```

These band structures are `BandStructure` objects (documented here). Among other things, this class implements a `plot` function that allows us to plot them on the same set of axes:

```
# Plot the two band structures
ax = lda_bs_shifted.plot(label='LDA', spin=0, color='tab:blue', ls='--')
ax = koopmans_bs_shifted.plot(ax=ax, label='K@LDA', color='tab:green')
```

With a few further aesthetic tweaks (download the full script `here`) we obtain the following plot:

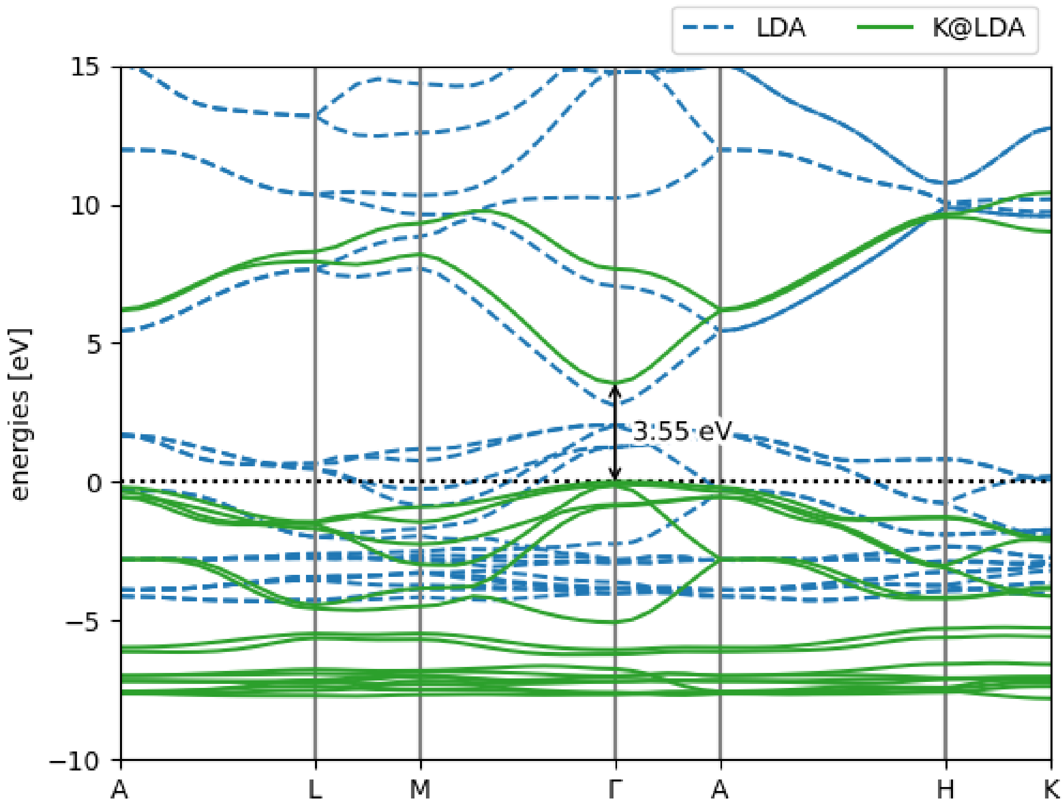
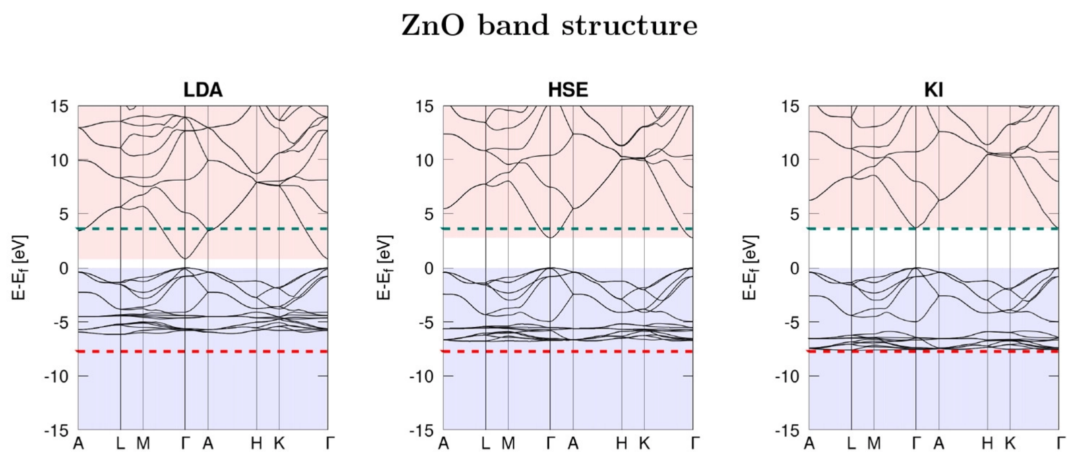


Fig. 3: Pretty band structure plot comparing the LDA and Koopmans band structures of ZnO

This script has also labelled the band gap for us, which is 3.55 eV. This is pretty close to the 3.62 eV result from Ref. [5] (see the below figure). The slight discrepancy comes from the fact that this tutorial has used coarser parameters to make the calculations run quickly.



	LDA	HSE	GW_0	$scG\tilde{W}$	KI	Exp.
E_{gap}(eV)	0.79	2.79	3.0	3.2	3.62	$3.60^{(*)}$
$\langle\varepsilon_d\rangle$(eV)	-5.1	-6.1	-6.4	-6.7	-6.9	-7.5/-8.0

Fig. 4: The band structure of ZnO from Colonna et al. [5]

8.3.2 Bonus material: understanding the Wannier projections

The first step to any calculation on a periodic system is to obtain a good set of Wannier functions. These depend strongly on our choice of the projections, which (for the moment) we must specify manually.

In the above calculation we gave you some Wannier projections to use:

```
        "w90": {
            "projections": [
                [{"site": "Zn", "ang_mtm": "l=0"}],
                [{"site": "Zn", "ang_mtm": "l=1"}],
                [{"site": "O", "ang_mtm": "l=0"}],
                [{"site": "Zn", "ang_mtm": "l=2"},
                 {"site": "O", "ang_mtm": "l=1"}],
                [{"site": "Zn", "ang_mtm": "l=0"}]
            ],
```

But what if you need to come up with your own Wannier projections? In this bonus section we will explain how to work out Wannier projections for yourself.

Configuring the Wannierization

To determine a good choice for the Wannier projections, we can first calculate a projected density of states (pDOS). Take your input file and change the `task` from `singlepoint` to `dft_bands`, and then run `koopmans zno.json`. This will run the DFT bandstructure workflow, which will produce a directory called `dft_bands` that contains various files, including a `png` of the bandstructure and pDOS. If you look at this file, you will see that the DFT band structure of ZnO consists of several sets of bands, each well-separated in energy space. As the pDOS shows us, the filled bands correspond to zinc 3s, zinc 3p, oxygen 2s, and then zinc 3d hybridized with oxygen 2p. Meanwhile, the lowest empty bands correspond to Zn 4s bands.

A sensible choice for the occupied projectors is therefore

```
"w90": {
   "projections": [
       [{"site": "Zn", "ang_mtm": "l=0"}],
       [{"site": "Zn", "ang_mtm": "l=1"}],
       [{"site": "O", "ang_mtm": "l=0"}],
       [{"site": "Zn", "ang_mtm": "l=2"},
        {"site": "O", "ang_mtm": "l=1"}]
   ]
```

Here we will use of the block-Wannierization functionality to wannierize each block of bands separately. If we didn't do this, then the Wannierization procedure might mix orbitals from different blocks of bands (the algorithm minimizes the spatial spread without regard to the energies of the orbitals it is mixing). This sort of mixing between orbitals of very different energies is generally detrimental to the Wannierization and the resulting Koopmans band structure.

For the empty bands we want to obtain two bands corresponding to the Zn 4s orbitals. These must be disentangled from the rest of the empty bands, which is achieved via the following `Wannier90` keywords.

`dis_win_max`
: defines the upper bound of the disentanglement energy window. This window should entirely encompass the lowest two bands corresponding to our Zn 4s projectors. Consequently, it will inevitably include some weights from higher bands

`dis_froz_max`
: defines the upper bound of the frozen energy window. This window should be as large as possible while excluding any bands that do not correspond to our Zn 4s projectors

To determine good values for these keywords, we clearly need a more zoomed-in band structure than the default. We can obtain this via the `*.fig.pkl` files that `koopmans` generates. Here is a short code snippet that replots the band structure over a narrower energy range

```python
import pickle

import matplotlib.pyplot as plt

# Use the python library "pickle" to load the *.fig.pkl file
fig = pickle.load(open('dft_bands/zno_dftbands_bandstructure.fig.pkl', 'rb'))

# Rescale the y axes
fig.axes[0].set_ylim([-5, 15])

# Show/save the figure (uncomment as desired)
plt.savefig('zno_dftbands_bandstructure_rescaled.png')
# plt.show()
```

Based on this figure, choose values for these two keywords and add them to your input file in the w90 block. Append the Zn 4s projections to the list of projections, too.

Note: dis_froz_max and dis_win_max should **not** be provided relative to the valence band edge. Meanwhile the band structure plots have set the valence band edge to zero. Make sure to account for this by shifting the values of dis_froz_max and dis_win_max by 9.3 eV (the valence band edge energy; you can get this value yourself via `grep 'highest occupied level' dft_bands/scf.pwo`)

Note: When disentanglement keywords such as dis_win_max are provided, they will only be used during the Wannierization of the final block of projections

Testing the Wannierization

To test your wannierization, you can now switch to the wannierize task and once again run `koopmans zno.json`. This will generate a wannier directory as well as a band structure plot, this time with the interpolated band structure plotted on top of the explicitly evaluated band structure. Ideally, the interpolated band structure should lie on top of the explicit one. Play around with the values of dis_froz_max and dis_win_max until you are happy with the resulting band structure.

Hint: Instead of using the *.fig.pkl file to obtain a zoomed-in band structure, add a plotting block to your json input file to manually set the y-limits:

```
"plotting": {
  "Emin": -5.0
  "Emax": 15
}
```

8.4 Tutorial 4: running convergence tests

While koopmans is a package primarily oriented towards performing Koopmans functional calculations, it does have a couple of other useful functionalities. Among these functionalities is the ability to perform arbitrary covnergence tests.

8.4.1 A simple convergence test

In this tutorial, we will make use of its convergence task to determine how large a cell size and energy cutoff is required to converge the PBE energy of the highest occupied molecular orbital (HOMO) of a water molecule. This workflow was chosen for its simplicity; it is possible to run convergence tests on any workflow implemented in the koopmans package.

The input file

In order to run this calculation, in the `workflow` block we need to set the `converge` parameter to true:

```
{
    "workflow": {
        "functional": "dft",
        "task": "singlepoint",
        "from_scratch": true,
        "converge": true,
        "fix_spin_contamination": false,
        "pseudo_library": "sg15_v1.0"
    },
```

and then provide the convergence information in the `convergence` block:

```
    "convergence": {
        "observable": "homo energy",
        "threshold": "0.01 eV",
        "variables": [
            "ecutwfc",
            "celldm1"
        ]
    },
```

These settings state that we are going to converge the HOMO energy to within 0.01 eV, with respect to *both* the energy cutoff `ecutwfc` and the cell size. The full input file can be found `here`.

The output file

When you run the calculation, you should see something like this after the header:

```
ecutwfc = 20.0, celldm1 = 11.3
------------------------------
   Running ./dft... done

ecutwfc = 20.0, celldm1 = 12.3
------------------------------
   Running ./dft... done

ecutwfc = 20.0, celldm1 = 13.3
------------------------------
   Running ./dft... done

ecutwfc = 30.0, celldm1 = 11.3
------------------------------
   Running ./dft... done

ecutwfc = 30.0, celldm1 = 12.3
------------------------------
   Running ./dft... done
```

Here, the code is attempting to use progressively larger energy cutoffs and cell sizes. It will ultimately arrive at a converged solution, with a `ecutwfc` of 50.0 Ha and a cell slightly larger than that provided in the `.json` input file.

Plotting

The individual `Quantum ESPRESSO` calculations reside in nested subdirectories. If you plot the HOMO energies from each of these, you should get something like this:

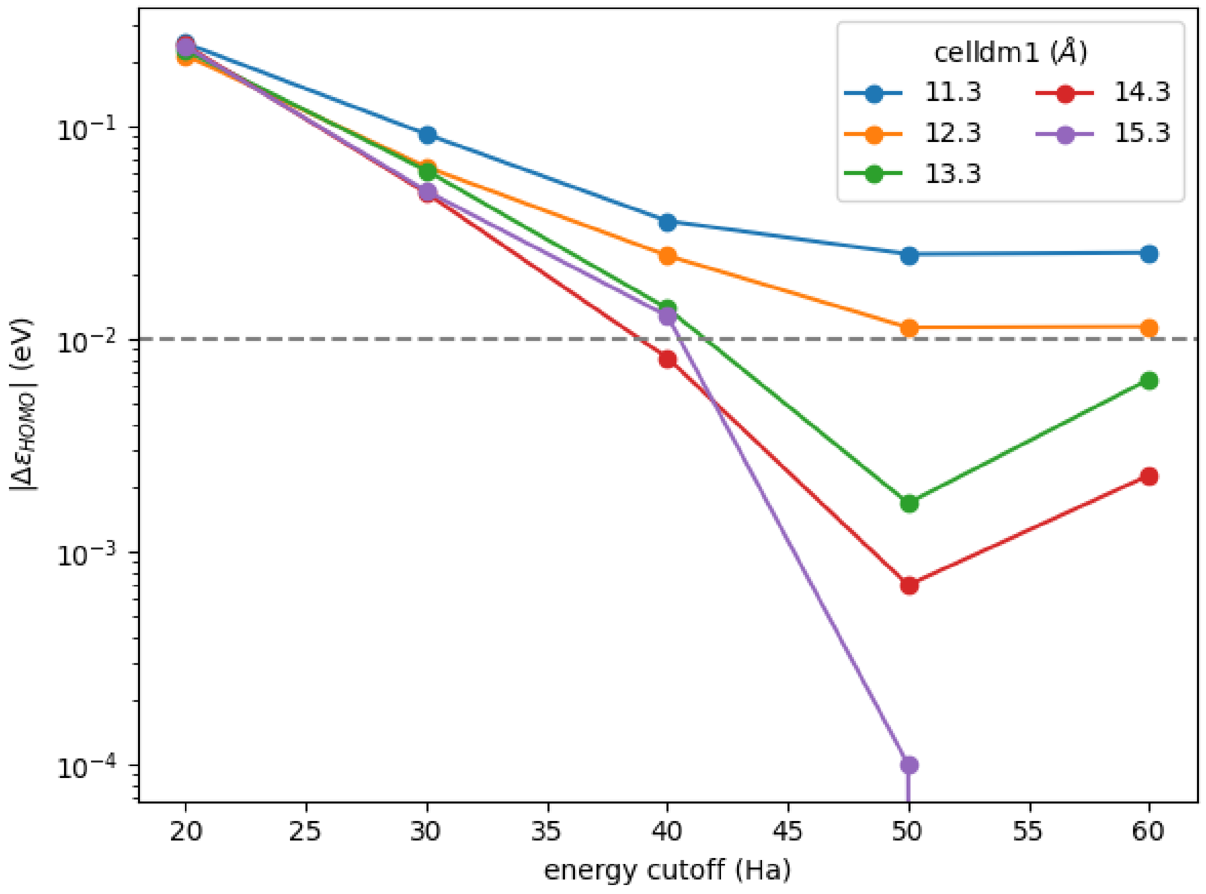


Fig. 5: Plot of the HOMO energy of water with respect to the energy cutoff and cell size (generated using `this script`)

We can see that indeed the calculation with `ecutwfc = 50.0` and `celldm(1) = 13.3` is the one where the energy of the HOMO goes within (and stays within) 0.01 eV of the most accurate calculation.

8.4.2 A custom convergence test

In the previous example, we performed a convergence test with respect to `ecutwfc` and `celldm1`. A full list of supported convergence variables can be found *here*. You will see that only a couple of variables are implemented by default. But don't let that limit you! It is possible to perform a convergence test on arbitrary keywords using factories.

First, try taking the input file from the first part of the tutorial, and switch the pseudopotential library to `pseudo_dojo_standard`. What do you notice?

Hopefully, the first thing you will see is that there are now some warnings about the small box parameters `nrb` like this,

```
UserWarning: Small box parameters "nrb" not provided in input: these will be␣
↪automatically set to safe default values. These values can probably be decreased, but␣
```

(continues on next page)

(continued from previous page)

```
↪this would require convergence tests.
Estimated real mesh dimension (nr1, nr2, nr3) = ...
Small box mesh dimension (nr1b, nr2b, nr3b) = ...
```

These parameters are associated with the way `Quantum ESPRESSO` handles non-local core corrections in pseudopotentials, and corrections are present in the new set of pseudopotentials but absent in the SG15 pseudopotentials.

So, let's perform a convergence test! The `nrb` parameters are not directly implemented as a convergence variable in `koopmans`, but we can use a factory to perform a convergence test on them, by making use of the `ConvergenceVariable` and `ConvergenceWorkflowfactory` classes.

```python
'''
A simple script that converges the HOMO energy of a water molecule with respect to nr1b,␣
↪nr2b, and nr3b
'''

from ase.build import molecule
from koopmans import workflows

# Use ASE to construct a water molecule
atoms = molecule('H2O', vacuum=5.0)

# Create a subworkflow which calculates (among other things) the PBE HOMO energy of water
subworkflow = workflows.DFTCPWorkflow(atoms=atoms, ecutwfc=30.0, base_functional='pbe',
                                      pseudo_library='pseudo_dojo_standard',
                                      calculator_parameters={'kcp': {'nr1b': 6, 'nr2b':␣
↪6, 'nr3b': 6}})

# koopmans doesn't implement convergence with respect to nrb, so we need to define a␣
↪custom
# ConvergenceVariable. To do so, we must first...
# ... define a function that extracts nr1-3b from a workflow
def get_nrb(workflow):
    return [workflows.get_calculator_parameter(workflow, f'nr{i}b') for i in [1, 2, 3]]

# ... define a function that sets nr1-3b
def set_nrb(workflow, value):
    workflows.set_calculator_parameter(workflow, 'nr1b', value[0])
    workflows.set_calculator_parameter(workflow, 'nr2b', value[1])
    workflows.set_calculator_parameter(workflow, 'nr3b', value[2])

nrb_variable = workflows.ConvergenceVariable(name='nrb',
                                             increment=[2, 2, 2],
                                             get_value=get_nrb,
                                             set_value=set_nrb)

# Create the convergence workflow using the convergence factory. Because koopmans knows␣
↪how to
# converge with respect to ecutwfc, we don't need to implement a custom␣
↪ConvergenceVariable for it
# and instead can just tell it the variable name
workflow = workflows.ConvergenceWorkflowFactory(subworkflow,
                                                observable='total energy',
```

(continues on next page)

(continued from previous page)

```
                                                    threshold=1e-3,
                                                    variables=[nrb_variable])

# Run the workflow
workflow.run()
```

Running this script will perform a convergence test with respect to `nrb 1-3`.

Warning: This tutorial performs convergence tests in a slightly incorrect way, To see this, add the keyword `length = 10` to the `ConvergenceVarialbe` in the above script. How is the behaviour different? Which behaviour is correct? Why?

8.5 Tutorial 5: using machine learning to predict the screening parameters of water molecules

In this tutorial, we will train a machine-learning model to predict the screening parameters of water molecules directly from their orbital densities. To generate a trajectory with 20 different atomic configurations, we run a `python script` that applies random noise to the atomic positions of a water molecule. The resulting atomic positions are saved in a `xyz file` and are visualized below

Our goal in this tutorial is to perform Koopmans calculations on each of these 20 snapshots using a machine learning model to predict the screening parameters instead of calculating them ab initio.

8.5.1 Running a machine learning workflow

To predict the screening parameters with the machine learning model we must first train the model. In the following we will use the first five snapshots for training and then use the trained machine learning model to predict the screening parameters for the remaining 15 snapshots.

The input file for the machine learning workflow

The input file for performing this task can be downloaded `here`.

First, we have to specify that we want to perform Koopmans calculations on a whole trajectory of snapshots by setting the `"task"` keyword in the `"workflow"` block:

```
    "workflow": {
        "task": "trajectory",
        "functional": "ki",
```

For this task, we don't provide the `"atomic_positions"` directly to the input file since we don't want to perform a Koopmans calculation on a single snapshot but on many snapshots. Instead, we provide an xyz file containing all the atomic positions of each snapshot that we would like to simulate

```
    "atoms": {
        "atomic_positions": {
```

(continues on next page)

(continued from previous page)

```
            "units": "angstrom",
            "snapshots": "snapshots.xyz"
```

Finally, we have to provide a `ml` block with keywords specific to the machine learning model

```
    "ml": {
        "use_ml": true,
        "n_max": 6,
        "l_max": 6,
        "r_min": 1.0,
        "r_max": 4.0,
        "number_of_training_snapshots": 5
    },
```

To predict the screening parameters from the orbital densities, we have to translate the orbital densities into input vectors for the machine learning model. To do so, we decompose the orbital densities into radial basis functions $g_{nl}(r)$ and angular basis functions $Y_{ml}(\theta, \phi)$. This decomposition has the following four hyperparameters that we provided in the input file:

- n_{max} determines the number of radial basis functions
- l_{max} determines the number of angular basis functions
- r_{min} determines the smallest cutoff radius for the radial basis functions
- r_{max} determines the largest cutoff radius for the radial basis functions

In anticipation that the machine learning model will be most useful in extended systems (liquids or solids), we apply periodic boundary conditions and use maximally localized Wannier functions as our variational orbitals (despite the fact that our toy water model is not, in fact, a periodic system).

The output file for the machine learning workflow

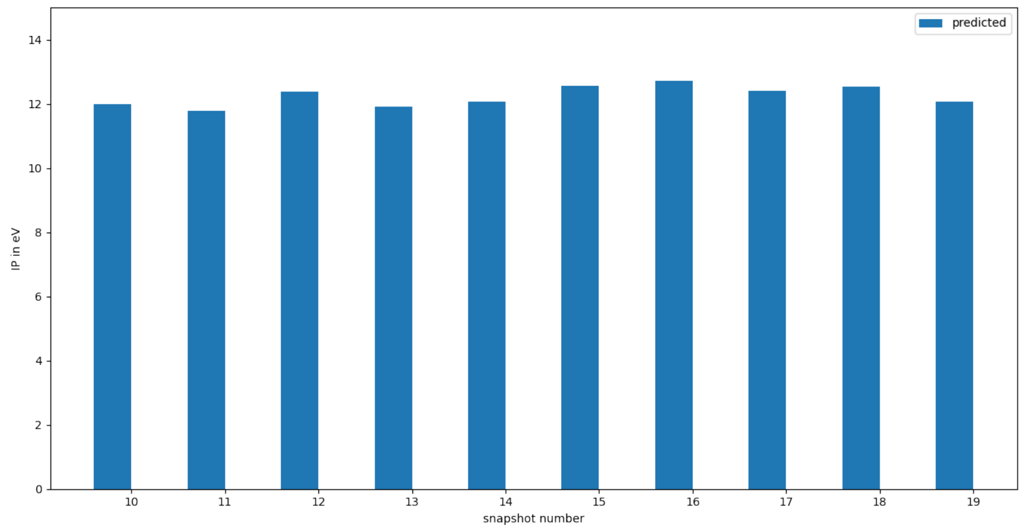
Running this calculation, the output will show that we compute the screening parameters of the first five snapshots ab initio and add the results to our training data

```
 Orbital 1
 ---------
  Running calc_alpha/orbital_1/dft_n-1... done

   predicted screening parameter:  1.00000
   calculated screening parameter: 0.57472
   absolute error:                 0.42528

   Adding this orbital to the training data
```

Then we use the trained model to predict the screening parameters of the remaining snapshots

```
 Orbital 1
 ---------
   Predicting the screening parameter with the ML model
```

Using the script `plot_5a.py` we can plot predicted ionization potentials of the water molecule across the last 10 snapshots. Of course, they don't necessarily correspond to anything physical because these configurations have been randomly generated. But in real applications the snapshots will correspond to something physical and the resulting ionization potentials will be meaningful.



Here there is no way of telling if the model is correct – it has provided us with some screening parameters and we have to trust it. If we want to check if a machine learning model is working properly what we need to do is a convergence analysis with respect to the number of training data. This will be the goal of the following section.

8.5.2 Running a convergence analysis

The input file for the convergence analysis

The corresponding `input file` differs from the previous input file only in the `"task"` keyword

```
    "workflow": {
        "task": "convergence_ml",
        "functional": "ki",
```

and the `"number_of_training_snapshots"`

```
    "ml": {
        "use_ml": true,
        "n_max": 6,
        "l_max": 6,
        "r_min": 1.0,
        "r_max": 4.0,
        "number_of_training_snapshots": 10,
        "quantities_of_interest": ["alphas", "evs"]
    },
    "atoms": {
        "atomic_positions": {
            "units": "angstrom",
```

For the `convergence_ml` task, setting `"number_of_training_snapshots": 10` means that we will perform the convergence analysis with respect to 1, 2, … , and 10 training snapshots and use the remaining snapshots (in this case snapshots 11 to 20) for testing.

The `"quantities_of_interest"` is the list of parameters with respect to which we would like to perform the convergence analysis. In addition to performing it only with respect to the screening parameters `"alphas"`, we also perform it with respect to the eigenvalues (`"evs"`). The latter requires an additional `final calculation` for each snapshot and therefore takes slightly longer to run.

The output file for the convergence analysis

You should see that the workflow first computes the screening parameters ab-initio for the last 10 snapshots.

```
Obtaining ab-initio results for the last 10 snapshot(s)
=======================================================
```

Next, snapshot 1 is added to the training data.

```
Adding snapshot 1 to the training data
======================================
```

After having trained the machine learning model on the orbitals of the first snapshot we use the trained model to predict the screening parameters of the last 10 snapshots and compare our results to the results from the ab initio computation.

```
Testing on the last 10 snapshot(s)
==================================
```

Next, we add snapshot 2 to the training data.

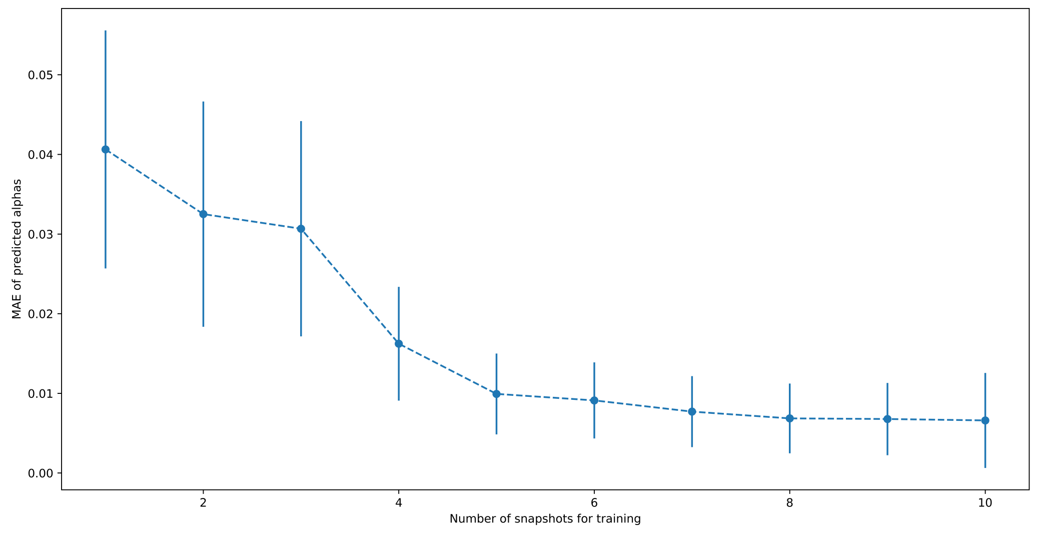
```
Adding snapshot 2 to the training data
======================================
```

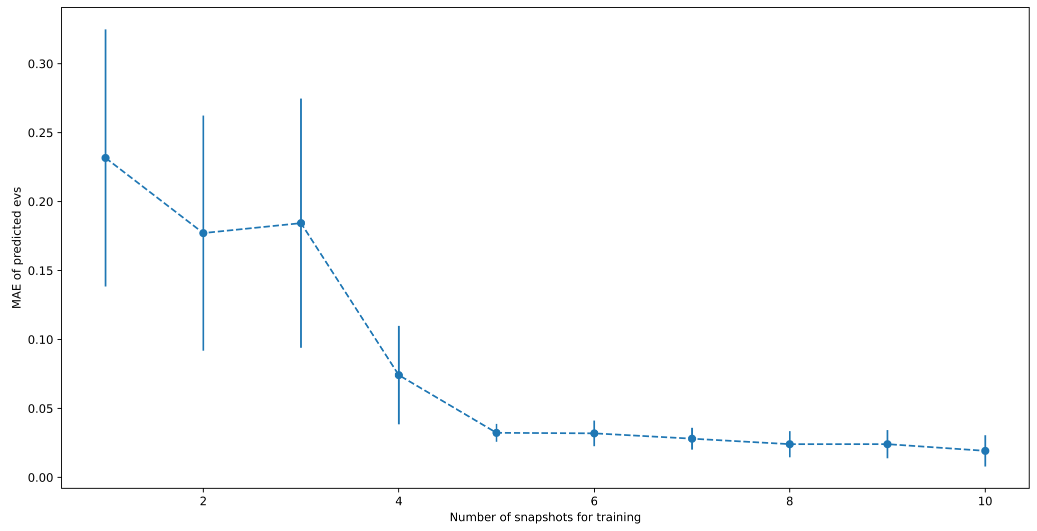
Our model is now trained on the orbitals of 2 snapshots. We use this model again to predict the screening parameters of the last 10 snapshots and compare the results to the ab initio calculation. We repeat this procedure until we have added all 10 snapshots to the training data. Then we can have a look at the convergence of the mean absolute error of the predicted screening parameters:

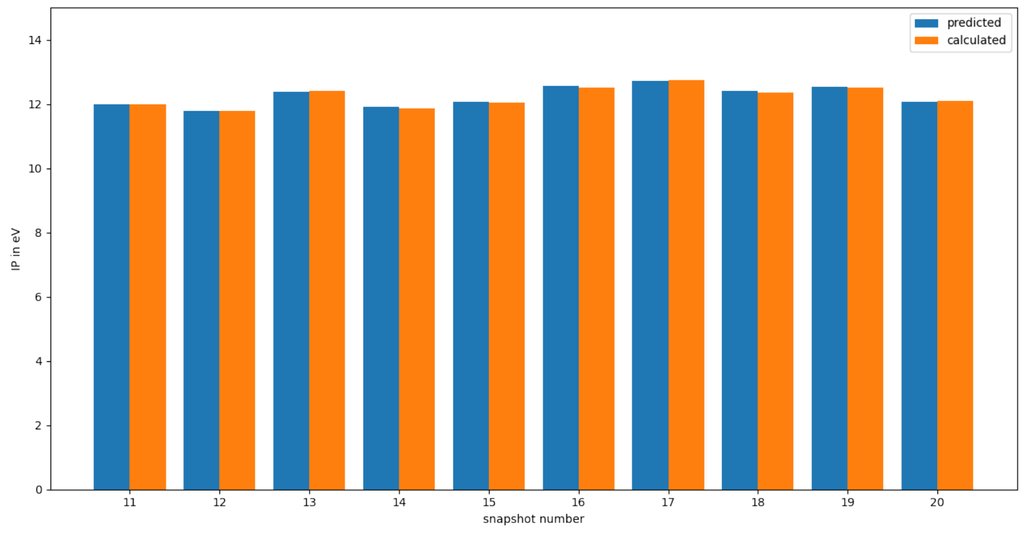
and the convergence of the mean absolute error of the predicted orbital energies:

(You can find these plots in the `convergence_analysis/final_results/` subdirectory.) We can see that we converged to a reasonable accuracy after about 5 training snapshots (which corresponds to 20 occupied and 10 empty orbitals).

We can now also check (`plot_5b.py`) that the predicted ionization potentials match with the ionization potentials obtained from the ab-initio computation of the screening parameters:


MAE of predicted alphas
Number of snapshots for training
0.00
0.01
0.02
0.03
0.04
0.05
2
4
6
8
10


MAE of predicted evs
Number of snapshots for training
0.00
0.05
0.10
0.15
0.20
0.25
0.30
2
4
6
8
10


predicted
calculated
IP in eV
snapshot number
0
2
4
6
8
10
12
14
11
12
13
14
15
16
17
18
19
20

CHAPTER

NINE

SUPPORT AND FEEDBACK

If you have any questions about running `koopmans`, post it to our mailing list

If you have a bug to report, please open an issue on GitHub

CHAPTER

TEN

REFERENCES

10.1 Selected references

In any publication arising from the use of Koopmans functionals and/or the `koopmans` code, please cite

- E. B. Linscott, N. Colonna, R. De Gennaro, N. L. Nguyen, G. Borghi, A. Ferretti, I. Dabo, and N. Marzari. Koopmans: An Open-Source Package for Accurately and Efficiently Predicting Spectral Properties with Koopmans Functionals. *J. Chem. Theory Comput.*, August 2023. doi:10.1021/acs.jctc.3c00652.

Other relevant references include

Papers introducing Koopmans functionals

- I. Dabo, M. Cococcioni, and N. Marzari. Non-Koopmans Corrections in Density-functional Theory: Self-interaction Revisited. January 2009. arXiv:0901.2637.
- I. Dabo, A. Ferretti, N. Poilvert, Y. Li, N. Marzari, and M. Cococcioni. Koopmans' condition for density-functional theory. *Phys. Rev. B*, 82(11):115121, September 2010. doi:10.1103/PhysRevB.82.115121.
- G. Borghi, A. Ferretti, N. L. Nguyen, I. Dabo, and N. Marzari. Koopmans-compliant functionals and their performance against reference molecular data. *Phys. Rev. B*, 90(7):075135, August 2014. doi:10.1103/PhysRevB.90.075135.
- G. Borghi, C. H. Park, N. L. Nguyen, A. Ferretti, and N. Marzari. Variational minimization of orbital-density-dependent functionals. *Phys. Rev. B*, 91(15):155112, April 2015. doi:10.1103/PhysRevB.91.155112.

Linear-response formalism

- N. Colonna, N. L. Nguyen, A. Ferretti, and N. Marzari. Screening in Orbital-Density-Dependent Functionals. *J. Chem. Theory Comput.*, 14(5):2549–2557, May 2018. doi:10.1021/acs.jctc.7b01116.

Application to molecules

- I. Dabo, A. Ferretti, C. H. Park, N. Poilvert, Y. Li, M. Cococcioni, and N. Marzari. Donor and acceptor levels of organic photovoltaic compounds from first principles. *Phys. Chem. Chem. Phys.*, 15(2):685–695, January 2013. doi:10.1039/c2cp43491a.
- N. L. Nguyen, G. Borghi, A. Ferretti, I. Dabo, and N. Marzari. First-Principles Photoemission Spectroscopy and Orbital Tomography in Molecules from Koopmans-Compliant Functionals. *Phys. Rev. Lett.*, 114(16):166405, April 2015. doi:10.1103/PhysRevLett.114.166405.
- N. L. Nguyen, G. Borghi, A. Ferretti, and N. Marzari. First-Principles Photoemission Spectroscopy of DNA and RNA Nucleobases from Koopmans-Compliant Functionals. *J. Chem. Theory Comput.*, 12(8):3948–3958, August 2016. doi:10.1021/acs.jctc.6b00145.
- N. Colonna, N. L. Nguyen, A. Ferretti, and N. Marzari. Koopmans-compliant functionals and potentials and their application to the GW100 test set. *J. Chem. Theory Comput.*, 15(3):1905–1914, March 2019. doi:10.1021/acs.jctc.8b00976.

Application to solids

- N. L. Nguyen, N. Colonna, A. Ferretti, and N. Marzari. Koopmans-compliant spectral functionals for extended systems. *Phys. Rev. X*, 8(2):021051, May 2018. doi:10.1103/PhysRevX.8.021051.
- R. De Gennaro, N. Colonna, E. Linscott, and N. Marzari. Bloch's theorem in orbital-density-dependent functionals: Band structures from Koopmans spectral functionals. *Phys. Rev. B*, 106(3):035106, July 2022. doi:10.1103/PhysRevB.106.035106.
- N. Colonna, R. D. Gennaro, E. Linscott, and N. Marzari. Koopmans Spectral Functionals in Periodic Boundary Conditions. *J. Chem. Theory Comput.*, August 2022. doi:10.1021/acs.jctc.2c00161.

Connection with many-body formulations

- A. Ferretti, I. Dabo, M. Cococcioni, and N. Marzari. Bridging density-functional and many-body perturbation theory: orbital-density dependence in electronic-structure functionals. *Phys. Rev. B*, 89(19):195134, May 2014. doi:10.1103/PhysRevB.89.195134.

10.2 All references

CHAPTER ELEVEN

USEFUL LINKS

11.1 koopmans

- Github
- Mailing list

11.2 Quantum Espresso

- Website
- Gitlab repository
- List of keywords for pw.x
- List of keywords for cp.x

11.3 EPFL

- THEOS group home page

BIBLIOGRAPHY

[1] V. I. Anisimov and A. V. Kozhevnikov. Transition state method and Wannier functions. *Phys. Rev. B*, 72(7):075125, August 2005. doi:10.1103/PhysRevB.72.075125.

[2] S. Baroni, S. de Gironcoli, A. Dal Corso, and P. Giannozzi. Phonons and related crystal properties from density-functional perturbation theory. *Rev. Mod. Phys.*, 73(2):515–562, July 2001. doi:10.1103/RevModPhys.73.515.

[3] G. Borghi, A. Ferretti, N. L. Nguyen, I. Dabo, and N. Marzari. Koopmans-compliant functionals and their performance against reference molecular data. *Phys. Rev. B*, 90(7):075135, August 2014. doi:10.1103/PhysRevB.90.075135.

[4] G. Borghi, C. H. Park, N. L. Nguyen, A. Ferretti, and N. Marzari. Variational minimization of orbital-density-dependent functionals. *Phys. Rev. B*, 91(15):155112, April 2015. doi:10.1103/PhysRevB.91.155112.

[5] N. Colonna, R. D. Gennaro, E. Linscott, and N. Marzari. Koopmans Spectral Functionals in Periodic Boundary Conditions. *J. Chem. Theory Comput.*, August 2022. doi:10.1021/acs.jctc.2c00161.

[6] N. Colonna, N. L. Nguyen, A. Ferretti, and N. Marzari. Screening in Orbital-Density-Dependent Functionals. *J. Chem. Theory Comput.*, 14(5):2549–2557, May 2018. doi:10.1021/acs.jctc.7b01116.

[7] N. Colonna, N. L. Nguyen, A. Ferretti, and N. Marzari. Koopmans-compliant functionals and potentials and their application to the GW100 test set. *J. Chem. Theory Comput.*, 15(3):1905–1914, March 2019. doi:10.1021/acs.jctc.8b00976.

[8] I. Dabo, M. Cococcioni, and N. Marzari. Non-Koopmans Corrections in Density-functional Theory: Self-interaction Revisited. January 2009. arXiv:0901.2637.

[9] I. Dabo, A. Ferretti, C. H. Park, N. Poilvert, Y. Li, M. Cococcioni, and N. Marzari. Donor and acceptor levels of organic photovoltaic compounds from first principles. *Phys. Chem. Chem. Phys.*, 15(2):685–695, January 2013. doi:10.1039/c2cp43491a.

[10] I. Dabo, A. Ferretti, N. Poilvert, Y. Li, N. Marzari, and M. Cococcioni. Koopmans' condition for density-functional theory. *Phys. Rev. B*, 82(11):115121, September 2010. doi:10.1103/PhysRevB.82.115121.

[11] R. De Gennaro, N. Colonna, E. Linscott, and N. Marzari. Bloch's theorem in orbital-density-dependent functionals: Band structures from Koopmans spectral functionals. *Phys. Rev. B*, 106(3):035106, July 2022. doi:10.1103/PhysRevB.106.035106.

[12] A. Ferretti, I. Dabo, M. Cococcioni, and N. Marzari. Bridging density-functional and many-body perturbation theory: orbital-density dependence in electronic-structure functionals. *Phys. Rev. B*, 89(19):195134, May 2014. doi:10.1103/PhysRevB.89.195134.

[13] D. R. Hamann. Optimized norm-conserving Vanderbilt pseudopotentials. *Phys. Rev. B*, 88(8):085117, August 2013. doi:10.1103/PhysRevB.88.085117.

[14] E. Kraisler and L. Kronik. Piecewise Linearity of Approximate Density Functionals Revisited: Implications for Frontier Orbital Energies. *Phys. Rev. Lett.*, 110(12):126403, March 2013. doi:10.1103/PhysRevLett.110.126403.

[15] L. Kronik, T. Stein, S. Refaely-Abramson, and R. Baer. Excitation Gaps of Finite-Sized Systems from Optimally Tuned Range-Separated Hybrid Functionals. *J. Chem. Theory Comput.*, 8(5):1515–1531, May 2012. doi:10.1021/ct2009363.

[16] K. Lejaeghere, G. Bihlmayer, T. Björkman, P. Blaha, S. Blügel, V. Blum, D. Caliste, I. E. Castelli, S. J. Clark, A. Dal Corso, S. de Gironcoli, T. Deutsch, J. K. Dewhurst, I. Di Marco, C. Draxl, M. Dułak, O. Eriksson, J. A. Flores-Livas, K. F. Garrity, L. Genovese, P. Giannozzi, M. Giantomassi, S. Goedecker, X. Gonze, O. Grånäs, E. K. U. Gross, A. Gulans, F. Gygi, D. R. Hamann, P. J. Hasnip, N. A. W. Holzwarth, D. Iuşan, D. B. Jochym, F. Jollet, D. Jones, G. Kresse, K. Koepernik, E. Küçükbenli, Y. O. Kvashnin, I. L. M. Locht, S. Lubeck, M. Marsman, N. Marzari, U. Nitzsche, L. Nordström, T. Ozaki, L. Paulatto, C. J. Pickard, W. Poelmans, M. I. J. Probert, K. Refson, M. Richter, G.-M. Rignanese, S. Saha, M. Scheffler, M. Schlipf, K. Schwarz, S. Sharma, F. Tavazza, P. Thunström, A. Tkatchenko, M. Torrent, D. Vanderbilt, M. J. van Setten, V. Van Speybroeck, J. M. Wills, J. R. Yates, G.-X. Zhang, and S. Cottenier. Reproducibility in density functional theory calculations of solids. *Science*, 351(6280):aad3000, March 2016. doi:10.1126/science.aad3000.

[17] C. Li, X. Zheng, N. Q. Su, and W. Yang. Localized orbital scaling correction for systematic elimination of delocalization error in density functional approximations. *Natl. Sci. Rev.*, 5:203–215, 2018. URL: https://academic.oup.com/nsr/article/5/2/203/4104965 (visited on 2020-04-04).

[18] E. B. Linscott, N. Colonna, R. De Gennaro, N. L. Nguyen, G. Borghi, A. Ferretti, I. Dabo, and N. Marzari. Koopmans: An Open-Source Package for Accurately and Efficiently Predicting Spectral Properties with Koopmans Functionals. *J. Chem. Theory Comput.*, August 2023. doi:10.1021/acs.jctc.3c00652.

[19] J. Ma and L.-W. Wang. Using Wannier functions to improve solid band gap predictions in density functional theory. *Sci. Rep.*, 6(1):24924, April 2016. doi:10.1038/srep24924.

[20] N. Marzari, A. A. Mostofi, J. R. Yates, I. Souza, and D. Vanderbilt. Maximally localized Wannier functions: Theory and applications. *Rev. Mod. Phys.*, 84(4):1419–1475, October 2012. doi:10.1103/RevModPhys.84.1419.

[21] F. Nattino, C. Dupont, N. Marzari, and O. Andreussi. Functional Extrapolations to Tame Unbound Anions in Density-Functional Theory Calculations. *J. Chem. Theory Comput.*, 15(11):6313–6322, November 2019. doi:10.1021/acs.jctc.9b00552.

[22] N. L. Nguyen, G. Borghi, A. Ferretti, I. Dabo, and N. Marzari. First-Principles Photoemission Spectroscopy and Orbital Tomography in Molecules from Koopmans-Compliant Functionals. *Phys. Rev. Lett.*, 114(16):166405, April 2015. doi:10.1103/PhysRevLett.114.166405.

[23] N. L. Nguyen, G. Borghi, A. Ferretti, and N. Marzari. First-Principles Photoemission Spectroscopy of DNA and RNA Nucleobases from Koopmans-Compliant Functionals. *J. Chem. Theory Comput.*, 12(8):3948–3958, August 2016. doi:10.1021/acs.jctc.6b00145.

[24] N. L. Nguyen, N. Colonna, A. Ferretti, and N. Marzari. Koopmans-compliant spectral functionals for extended systems. *Phys. Rev. X*, 8(2):021051, May 2018. doi:10.1103/PhysRevX.8.021051.

[25] M. R. Pederson, R. A. Heaton, and C. C. Lin. Local-density Hartree–Fock theory of electronic states of molecules with self-interaction correction. *J. Chem. Phys.*, 80(5):1972–1975, March 1984. doi:10.1063/1.446959.

[26] G. Prandini, A. Marrazzo, I. E. Castelli, N. Mounet, and N. Marzari. Precision and efficiency in solid-state pseudopotential calculations. *npj Comput Mater*, 4(1):1–13, December 2018. doi:10.1038/s41524-018-0127-2.

[27] P. Scherpelz, M. Govoni, I. Hamada, and G. Galli. Implementation and Validation of Fully Relativistic GW Calculations: Spin–Orbit Coupling in Molecules, Nanocrystals, and Solids. *J. Chem. Theory Comput.*, 12(8):3523–3544, August 2016. doi:10.1021/acs.jctc.6b00114.

[28] M. Schlipf and F. Gygi. Optimization algorithm for the generation of ONCV pseudopotentials. *Computer Physics Communications*, 196:36–44, November 2015. doi:10.1016/j.cpc.2015.05.011.

[29] Y. Schubert, N. Marzari, and E. Linscott. Testing Koopmans spectral functionals on the analytically solvable Hooke's atom. *The Journal of Chemical Physics*, 158(14):144113, April 2023. doi:10.1063/5.0138610.

[30] J. H. Skone, M. Govoni, and G. Galli. Nonempirical range-separated hybrid functionals for solids and molecules. *Phys. Rev. B*, 93(23):235106, June 2016. doi:10.1103/PhysRevB.93.235106.

[31] D. Wing, G. Ohad, J. B. Haber, M. R. Filip, S. E. Gant, J. B. Neaton, and L. Kronik. Band gaps of crystalline solids from Wannier-localization–based optimal tuning of a screened range-separated hybrid functional. *Proc. Natl. Acad. Sci.*, 118(34):e2104556118, August 2021. doi:10.1073/pnas.2104556118.

[32] M. J. van Setten, M. Giantomassi, E. Bousquet, M. J. Verstraete, D. R. Hamann, X. Gonze, and G. -M. Rignanese. The PseudoDojo: Training and grading a 85 element optimized norm-conserving pseudopotential table. *Computer Physics Communications*, 226:39–54, May 2018. doi:10.1016/j.cpc.2018.01.012.

INDEX