Simulation Science Laboratory 2018

An Analysis Tool for Materials Design

Students:
Praneeth Katta Venkatesh Babu
Christian Partmann
Johannes Wasmer

Supervisors:
Stefan Blügel Prof. Dr.
Stefan Rost MSc
Quantum Theory of Materials PGI-1
Forschungszentrum Jülich

Abstract — The electronic band structure and the density of states of a material can be used to study many of its electrical, magnetic and optical properties. Band structures of real-world materials are quite complex and not accessible with analytic methods. One way to obtain them is numerical simulation using density functional theory (DFT) on high-performance computers. Still, the raw numerical data output by itself is not easy to interpret. Interactive visualization as a post-processing step can aid to explore that space and find the important features. In this work we develop a freely available post-processing pipeline for band structure calculations of Fleur, a realization of DFT based on the all-electron full potential linearized augmented planewave method. It features a generalized Python interface for processing and visualizing data in the hierarchical data format HDF5. Two graphical user interfaces (GUIs) built on that interface are introduced. Interactive controls let the user visually distinguish the effects of different atom groups, orbitals and defect states in supercells on the band structure. Simulations of $\mathrm{MoSe_2}$ and Co materials are examined using those GUIs, helping to deduce some of their physical properties.

Keywords band structure, density of states, visualization, DFT simulation, Fleur, HDF5

AICES
Schinkelstr. 2
Rogowski Building
4th Floor
52062 Aachen

Acknowledgments

The authors would like to thank the following members of the section Quantum Theory of Materials in the Peter-Grünberg Institute at the Forschungszentrum Jülich: our supervisors Prof. Dr. Stefan Blügel and Stefan Rost for their helpful input, their time and commitment; Dr. Gregor Michalicek and Jens Bröder for their inspiring ideas.

Contents

List of Figures

Chapter 1

Introduction

Problem Statement

An important problem in solid-state physics is the computation of materials properties. Since the quantum mechanical equations which describe the physics of electrons in solids are usually impossible to solve with analytical methods, numerical codes were developed to solve these equations efficiently on high-performance computers. The project 'Fleur' that this project is based on is a highly optimized code that solves the many-body problem based on the DFT (Density functional theory) approach [Blü+18].

Fleur is a freely available all-electron full potential linearized augmented planewave (FLAPW) code developed at the Forschungzentrum Jülich. Like any other kind of numerical simulation, DFT simulations produce a significant amount of data that needs to be preprocessed, visualized and analyzed in order to gain physical insight. Since Walter Kohn's Nobel Prize for the development of DFT in 1998 [Koh99], supercomputers have become roughly a hundred thousand times faster[Meu+14], and the improvement of methods and algorithms has steadily kept pace with that evolution. Especially in the field of materials design, DFT simulation today is a manifestation of high-throughput computing, meaning that manual data processing becomes impractical. It is therefore imperative to supply practitioners with tools that automate as much of that repetitive part of the workload as efficiently as possible, leaving more room for the scientific endeavor.

The whole pipeline, from the raw data generated by Fleur through data exploration to comprehensible plots showing selected physical properties of the simulated material, is addressed in this project.

Motivation and Requirements

The main project goal was to develop a software product that is able to perform all necessary steps of postprocessing including visualization in order to make the Fleur simulation output easily accessible for both solid-state physicists and non-experts who use Fleur for the first time alike.

This imposes several requirements on the software:

- Physically accurate and meaningful representation of data: The data reduction strategy during the preprocessing must be transparent and physically motivated. Plots should be in a format well known by any physicist.
- Easy to use: The tool must not be overloaded with functionalities for very specific problems, but is supposed to be general and usable in an intuitive way without much explanation.
- Easy to access: The tool should be able to run in, or from, different environments with no additional setup.
- Reasonably fast: The datasets can be large, therefore efficient preprocessing and plotting techniques are necessary.
- Easy to extend and maintain: In order to let the software developed in the scope of this project keep pace with external improvements and new tools, the code should be written in a modular style.
- Export feature: Visualization results have to be exportable as PDF or PNG.

The central piece of the project from the physics point of view is the 'backend', which is a library that reads the simulation data and prepares it for the visualization. In this module, the dimensionality of data gets reduced according to the choice of the user in a physically meaningful way. The preprocessed data is then passed to a 'frontend', that visualizes the preprocessed data. To reach a wide range of potential users, the authors decided to develop a graphical user interface (GUI), where parameters relevant to the visualization can be passed interactively. The visualization produced within this interactive tool can then be exported in order to be shared and discussed.

Project Steps

The project was organized into several steps with the supervisors.

- Understanding the problem: Understand the physics of the datasets and how the data is used in research. This leads to a list of requirements that the software has to fulfill in order to be usable in a productive way.

- Preprocessing: Write a backend that reads raw data and processes it by transforming it into a format that can be visualized in an efficient way. Therefore, the dimensionality of the data needs to be reduced without losing too much physical information.
- Exploring the data: Investigate possible challenges that might occur during the visualization of the data. (e.g. how to deal with points/datasets covering each other, ...)
- Visualization: The preprocessed data is visualized in a scatter plot with a format well known in the physics community.
- Frontend: A GUI with intuitive features is developed such that a wide range of users can access Fleur output without having to deal with the raw data.
- Test the usability of the software using typical Fleur output files. Prove that physical insight can be gained easily using the software.
- Deployment: Bring the project into a form that can be distributed.

Chapter 2

Theoretical Background

Fleur computes the electronic structure in crystals using the density functional theory approach (DFT), which is one of the state of the art method for this problem. The many-body Schrödinger equation, that can be used to describe electrons in solids, is almost impossible to solve directly. The reason is that the storing the wavefunctions of each of the N_e electrons in the system at each spatial gridpoint scales exponentially with N_e and exceeds the memory of any currently available computer for even quite small N_e. This motivates the DFT approach, that uses two fundamental theorems to reduce the computational complexity of the many-body problem significantly: The Hohenberg-Kohn theorem [HK64] allows to use the electron density instead of the N_e-electron wavefunctions to uniquely characterize the ground state of a system. With the Kohn-Sham equations [KS65], the interacting Hamiltonian of the system can be replaced by non-interacting equations with an effective potential. This so-called KS-DFT approach reduces the memory usage of the many-body problem on a grid from $\mathcal{O}(N_g^{3N_e})$ to $\mathcal{O}(N_g^3)$ and trades the interacting Hamiltonian for a set of non-interacting equations with a computational complexity of $\mathcal{O}(N_e^3)$ that have to be solved self-consistently. In general, the DFT approach is not just limited to computations of electrons in crystals, but is also for example used in chemistry to compute nonperiodic molecules. The output of a DFT calculation includes various physical quantities, for instance, the three-dimensional spatial electron density, which is the central quantity in DFT calculations. In this project, however, we focus on two quantities that are easy to interpret, already reduced in dimensionality and frequently used in both experimental and theoretical physics.

The band structure $E(\mathbf{k})$ represents the eigenenergies of the eigenfunctions of the Hamiltonian for each crystal momentum $\mathbf{k}$. It is the dispersion relation of electrons in the crystal and relates allowed momenta and energies. In general, $E(\mathbf{k})$ is defined at any point inside the Brillouin zone, which is a special choice of the unit cell in the reciprocal lattice of the crystal. Both, the real space lattice and the reciprocal lattice are shown for a face-centered cubic (fcc) crystal in figure 2.1. For larger unit cells with fewer symmetries, the Brillouin zone can be much more complicated than in the shown example. In order to reduce the dimensionality of $E(\mathbf{k})$ with

$\mathbf{k} \in \mathbb{R}^3$, the dispersion relation is only sampled along a discrete one-dimensional path between high symmetry points in the Brillouin zone. This path still contains most of the relevant physical features.

(a) Lattice in real space [Sch15]

(b) Brillouin zone [Ind08]

Figure 2.1: Brillouin zone of an fcc lattice. The red curve in the Brillouin zone represents a possible sampling path of $E(\mathbf{k})$ in the reciprocal lattice.

The second central quantity in this project is the density of states (DOS) $D(E)$, which describes the number of states per energy interval that is independent of the crystal momentum. In this sense, the DOS is also derived from $E(\mathbf{k})$, but instead of just selecting a subset, the $\mathbf{k}$ dependency is summed out. This sum is performed internally by Fleur, since $E(\mathbf{k})$ must be known at every grid point in the Brillouin zone. Both complementary quantities together are a common choice for comprehensive visualizations of electronic structure since they still capture most of the important physics.

For applications, it is useful to investigate where the contributions to $E(\mathbf{k})$ and $D(E)$ come from. Therefore, the data contains the weights of each basis function of the DFT calculation belonging to the individual atom groups and the orbitals. This means that spatial information about the system can be restored by considering only contributions from certain atom groups. (An atom group contains all atoms that are equivalent with respect to symmetries of the real space lattice.) On the other hand, the projection on the (hydrogen-type) orbitals s, p, d, f encode information about the shape of the wavefunction at each atom. These contributions are stored in the form of relative weights that can be summed to include contributions for multiple groups and orbitals. In case of distinct spins in the crystal, $E(\mathbf{k})$, $D(E)$ and the weights can be different for both spins and are therefore stored individually.

Chapter 3

Implementation

As per the requirements expounded upon in the introduction, the deliverable of the project should be a finished software product. The software is written in Python so as to integrate easily with the research group's ongoing software projects around the Fleur code [Blü+18]. These are chiefly the group's materials science tool collection `masci-tools` [RBR18] , where also this project's code is hosted, and the 'Automated Interactive Infrastructure and Database for Computational Science' (AiiDA) [Piz+16]. The product stakeholders split into frontend users and code developers. In order to accommodate this, the product is organized into three subpackages or -modules, see Figure 3.1a.

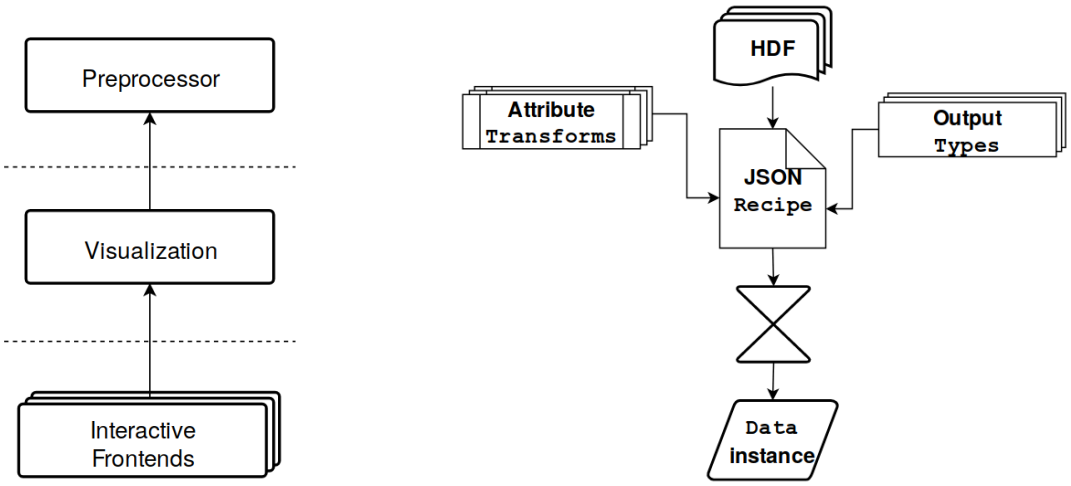
An important design consideration was to account for unknown use cases. This has been realized in each submodule by the decoupling of **interface** and **implementation**. The interfaces do not rely on any specific input file format, visualization method or library, unlike the implementations for a specific task or **application**. In this chapter, the word 'application' denotes the band structure and density of states visualization, and for these applications, implementations are provided.

This design choice was also one reason why the product does not reuse any of the `masci-tools` routines which partly solve quite similar problems, but seemed to be too specialized in a cursory code review. The project code aims to offer a more general framework into which some of these routines could be integrated.

HDF Preprocessor Module

Interface

This is the 'backend' of the tool. It is basically a file reader for the input data, for example a Fleur simulation output. Supported formats are the Hierarchical Data Format (HDF) [Kor11]



(a) Submodules (b) Preprocessor

Figure 3.1: Module Design. The arrows indicate dependency in a), and dataflow in b).

for the band structure, and a simple Fleur-specific comma-separated values (CSV) format for the density of states (DOS).

The HDF format is a flexible binary container for all kinds of common binary and text file formats. Each set of values or file of such a type constitutes a named `Dataset` inside the HDF file. The format supports metadata annotation and high-throughput input/output (I/O). As a consequence, it is considered by developers in some application domains that rely on numerical simulation codes, to arguably be one possible base for the establishment of common domain-specific rich data exchange standards in order to increase code interoperability. These developers are in the process of extending their codes' I/O capabilities towards that end. However, HDF's flexibility comes at the price of a relatively complex Application Programming Interface (API) as the keyhole for all operations.

The preprocessor module serves as a wrapper around that API by introducing the concept of `Recipes`, see Figure 3.1b. A specific application `Recipe` is a dictionary that aims to describe a complete Extract-Transform-Load (ETL) pipeline for one specific application of the original data. The 'extract' is the reading of a dataset from HDF, the 'transform' a sequence of once-through functions applied to the the dataset, and the 'load' the aggregation of all transformed datasets into one runtime object which has all the methods for operations on the data that are going to be used later on in the intended application.

The 'transform' and 'output' type methods are defined in hierarchical `Transform` and `Output_Type` classes. Multiple inheritance is used to sort them from general to application-specific applicability. This structure is built using Python's `AbstractBaseClass` (`ABC`). The advantages of the 'recipes approach' are:

- All ETL processes for one application are collected in one simple list (the recipe), not

locked in different code locations with conflicting contexts. In this list, entries can be sorted in any manner, e.g. alphabetical for perusal. Thus a recipe also serves as a concise documentation of how an application-domain HDF format should be handled.

- Recipes are de/serializable (can be read from and saved to disk) and thus be machine-created and manipulated, for instance in a workflow pipeline.
- The ETL processes, when declared in this interface, can be easily reused across applications. A recipe can combine different output types into a new type.

The feature that enables this flexibility is **type introspection**: the preprocessor processes the datasets listed in the recipe in the order of their mutual dependencies as found in the introspected listed transform and output methods. When all transformed datasets have been added to the object, all specified output types are searched and all their methods and attributes added. Thus the output object's type is defined at runtime, when the preprocessing is finished.

Implementation for Band Structure Visualization

The frontend has to draw three kinds of plots: a 3D atom plot of the unit cell or supercell, and a combined band structure and DOS plot sharing the same vertical energy axis. If no DOS data is present, the DOS part of the plot shall be omitted. All three plots are controlled by one set of graphical frontend control elements (widgets) for varying the parameters. In the current implementation, the data for the first two plots come from a HDF file, while the data for the DOS plot come from CSV files.

The band structure plot is a scatter plot. It plots discrete $E(\mathbf{k})$ data from the simulation. Firstly, the plotter needs the k-path (where $|\mathbf{k}| = k$) for the horizontal axis. The preprocessor, having received the recipe `FleurBands`, computes it from the k-points in the HDF in a transform. Secondly, the plotter needs the eigenergies for every point on the k-path, labeled by the band index ν, and its associated l-like charge $n_{s,k,\nu,g,l}$. This dataset is five-dimensional, and represents the contribution of spin s, k-point, band index ν, atom group g, and character or orbital l (here: only s,p,d,f) to the specific eigenergy. So plotting must involve a dimension reduction. The preprocessor resolves the processed data into the like-named output type `FleurBands`. This type has a data selection method. The respective `BandPlot` type calls this selector with a user selection of subsets of all (s, k, ν, g, l). The method then computes the according **effective weight** shown in Equation 3.1. The plotter uses that for the dot size of each $E(k)$ in the plot. Before rendering, the plotter normalizes the energies to the Fermi Level.

$$W^{\text{eff}}_{s,k,\nu} = \left(\frac{\sum\limits_{\substack{g \in \text{groups} \\ l \in \text{characters}}} n_{s,k,\nu,g,l}\, N_g}{\sum\limits_{\substack{g \in \text{all groups} \\ l \in \text{all characters}}} n_{s,k,\nu,g,l}\, N_g} \right) \left(W^{\text{unf}}_{s,k,\nu} \right)^{\alpha} \tag{3.1}$$

N_g denotes the number of atoms in a group. $W^{\text{unf}}_{s,k,\nu}$ is the unfolding weight, if the simulation was done with a supercell, and α its exponent. In the frontends, the latter is also a user control. The effect of unfolding is illustrated in Figure 3.2 for a toy example of a monatomic chain, with a two-atom supercell of size a' representing $\alpha = 0$ and a one-atom unit cell of size a representing $\alpha = 1$. A use-case is discussed in the Application Chapter 4.

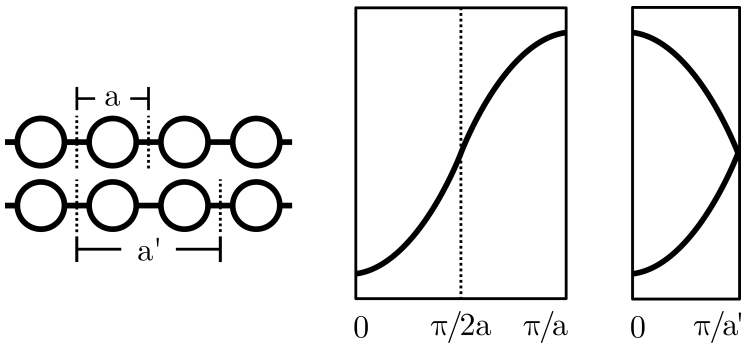


Figure 3.2: Band unfolding: example for a simple monatomic chain. Adapted from [Hof87].

Even for small band structures, the selector has to access on the order 10^7 individual data points for every selection change. Optimizations were introduced which included:

- a cutoff filter that skips effective weights too small to show up on the plot,
- use of optimized Numpy functions like `np.tensor` for the effective weight summation,
- buffering of unchanged data between two selections,
- array reshaping.

Together, these tweaks achieve a speedup of approximately 10^2 in plotting speed. Thanks to that, the tool remains usable even when the input HDF is in the 10^2 MB range.

Visualization Module & Interactive Graphical Frontends

Visualization Module

The Python visualization landscape abounds with a rapidly evolving plethora of plotting libraries for different application contexts and technology stacks [VR17]. Thus the project's visualization module's first design objective was to account for that fact by decoupling it from any specific library use, and modularizing it for intended applications. This structure again is built using Python's `AbstractBaseClass` (`ABC`) interface and multiple inheritance. Each application is represented by an abstract base class that contains the common plotting method signatures. Likewise, each plotting library is represented by an abstract base class that contains library-specifics. An *implementation* inherits both from one library base class and one or more

application base classes. See Fig. 3.3 for an impression. Thus switching the library in a use context should require minimal adjustment, and a new application (base class) can be be built from existing ones.

The second design objective was for the plotting methods to hide all interactions with the actual plotting library used under the hood. The application's base class' method arguments should only be tied to the data, not the plotting library. Thus different frontend implementations need no or minimal individual setup beforey they can call the same method for one specific plot and receive the identical visualization with identical interactive behavior.

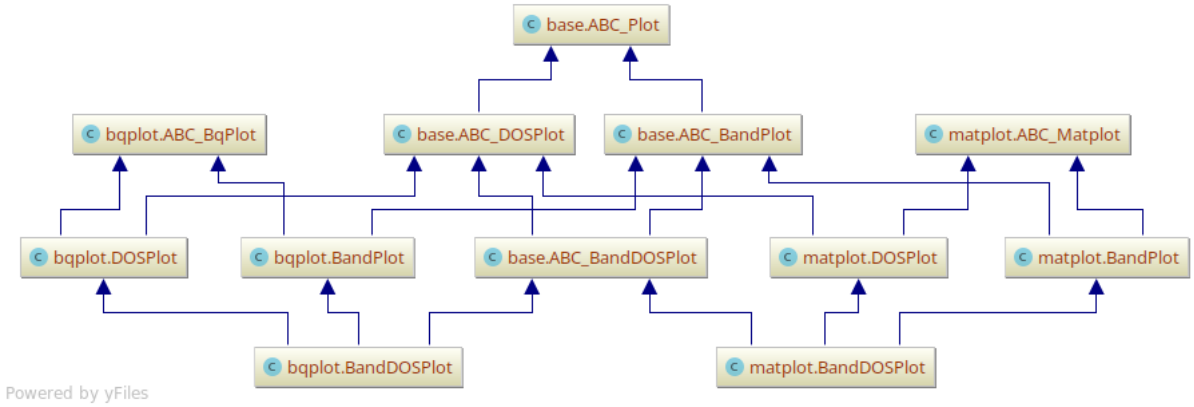


Figure 3.3: Visualization Module Design: Example inheritance for two applications `BandPlot` and `DOSPlot`, and two plotting libraries `matplotlib` and `bqplot`.

Desktop Frontend

The obvious advantage of a desktop-based GUI over a web-based tool obviously is that, as opposed to a web-based GUI that is probably hosted on a remote machine, input files that are stored locally do not have to be uploaded.

Since the reading of HDF files and preprocessing of the data is done using Python, it was decided that it would be best to also use Python for frontend development. Considering various packages for frontend in Python such as PyQt, Tkinter and other libraries available for desktop GUIs, Tkinter was selected, as it is bundled with Python and thus meets the requirement of easy maintainability best. It is a package where every button can be designed and can be assigned to a function. GUI types like `Label`, `Button`, `CheckButton`, `ListBox`, `Canvas` for plots, `Tab` for viewing each plot in different tab have been used to make a simple desktop frontend. It is simple to use with limited options for what the end-user needs. It is also easy to convert the desktop frontend code into a executable software and run in any system without any installation.

Web Frontend

As web frontends continue to replace traditional desktop frontends in many application domains [Dou+08], so Python-based frontends and visualizations are increasingly moving towards the browser, too. There, GUIs with interactive visualizations are often called **Dashboards**. For this project, a survey was undertaken to find the most suitable technology stack for a web frontend. The full survey is documented in [Was19]. The requirements for the solution, on top of those stated in the introduction, were defined as follows: The solution...

1. **openness**: relies solely on Open-Source-Software (OSS) with licensing suitable for academic use, has a stable release cycle, developer base and documentation,
2. **dashboarding**: features graphical control elements (widgets) that interact with InfoVis[1] plotting libraries,
3. **deployment**: ideally works like any web service, i.e. only a modern web browser is required to use it,
4. **maintenance**: requires only Python and no Web Development knowledge like e.g. Javascript, with respect to the product stakeholders.

The last point implies a client-server model where the dashboard app is hosted on a remote machine. This model requires a communication framework and protocol between the Python interpreter running on the server and the JavaScript interpreter running in the client browser. As per requirement no. 4, unlike a generic Python web framework like e.g. Flask, the framework should take care of that communication by itself. Four major frameworks were identified which fulfill the first three requirements: Project Jupyter, PyViz, Bokeh, and Dash by Plotly. The last two only partially fulfilled the requirement no. 4, so they were discarded. PyViz is the newest contender among these four. Its expressed goal is to untangle the Python visualization jungle by providing one high-level API that ties together all major Python InfoVis libraries and data formats, including support for dashboarding. This ambitious goal comes at the price of sacrificing support for 3D plotting [Aut18], which was needed in this project for the atoms plot. So PyViz had to be discarded.

This left Project Jupyter. By now, a wide variety of popular plotting libraries have made their tools capable of working with Jupyter's widget library `ipywidgets`. However, Jupyter only partially fulfills requirement no. 3 – a Jupyter notebook (app) cannot, by itself, be published (deployed) as a stand-alone website outside a live Jupyter environment [Bed18]:

> [...] "However, despite their web-based interactivity, the ipywidgets-based libraries (ipyleaflet, pythreejs, ipyvolume, bqplot) are difficult to deploy as public-facing apps because the Jupyter protocol allows arbitrary code execution" [...].

[1] InfoVis libraries: visualizations of information in arbitrary spaces, not necessarily the three-dimensional physical world. Example: matplotlib. SciVis libraries: visualizing physically situated data. Example: VTK [Bed18].

To avoid requiring users to setup a working Jupyter environment on their machine, the go-to solution for this problem is to setup a JupyterHub multi-user server. This still requires users to register an account there, so it's not completely open. Fortunately though, the intended users are contributors to the AiiDA project, and so should have access to the JupyterHub-based AiiDA Lab service where the app can be registered. Details on this procedure and alternative hosting solutions can be found in the developer section of the manual A on page 27.

Chapter 4

Applications

To illustrate the use of the graphical user interface, two different physical applications are shown in the desktop and the web frontend, respectively. From the physics point of view, the example in the desktop version focuses more on the density of states and the visualization of spin contributions, while the dataset in web frontend focuses on the band structure $E(\mathbf{k})$ and the visualization of defect states in supercells.

Web Frontend: $\mathrm{MoSe_2}$ Crystal

Figure 4.2 shows the visualization of a band structure calculation of a three-dimensional Molybdenum diselenide ($\mathrm{MoSe_2}$ bulk) crystal using the default settings of the GUI. Even with the default settings the band structure plots clearly indicate, that $\mathrm{MoSe_2}$ is a semiconductor since there are no states at the Fermi level. Because the minimum of the conduction band is located at another k as the maximum of the valence band, the plot shows that $\mathrm{MoSe_2}$ has an indirect band gap. This indicates that for the transition with the smallest energy difference between valence and the conduction band, energy and momentum have to change.

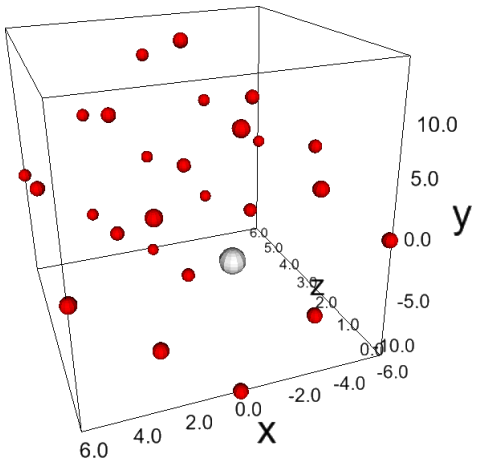


Figure 4.1: Atom plot of a $\mathrm{MoSe_2}$ monolayer with the defect atom selected in the web frontend

In contrast to the three-dimensional extended $\mathrm{MoSe_2}$ crystal, a $\mathrm{MoSe_2}$ monolayer (see Fig. 4.3) has a direct band gap but is still a semiconductor. Furthermore, the $\mathrm{MoSe_2}$ monolayer has a defect atom in every 9th unit cell and the DFT computation is therefore done in a 3×3 supercell to restore periodicity. This is the reason for the much greater number of states

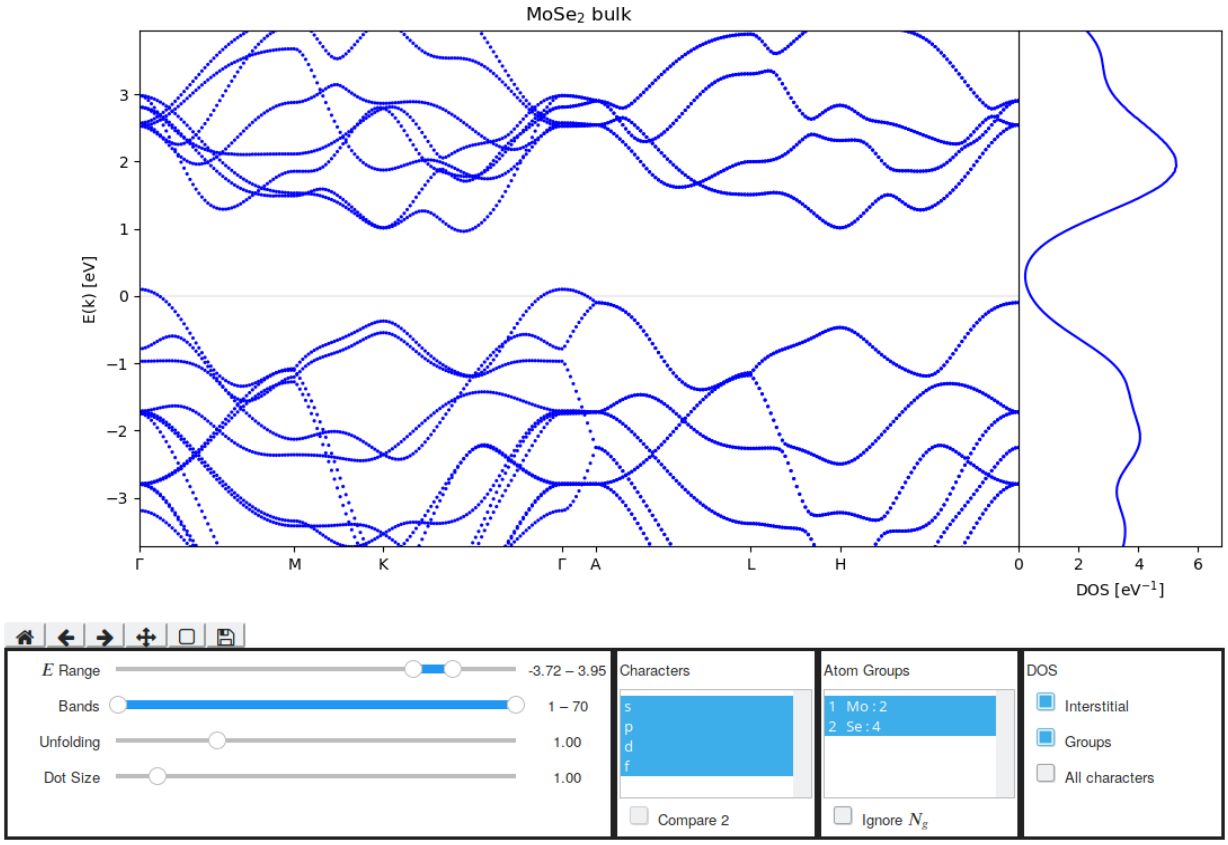


Figure 4.2: Band structure of a $MoSe_2$ crystal using the default settings of the web frontend. An indirect bandgap is visible.

in the band structure plot.

Since the Brillouin zone of the supercell is smaller than the Brillouin zone of the same crystal without the defect, the supercell Brillouin zone is unfolded to the same size as the Brillouin zone of the unperturbed lattice. To account for the fact that the defect is only present in every 9th cell and its relative importance for the spectrum is therefore degraded, an unfolding weight is introduced to visualize the relative importance of bands in the unfolded Brillouin zone. By default, the unfolding weight is used by our visualization tool, but it can gradually be turned off in order to highlight the impact of defect states. This is shown in figure 4.4. To even better visualize the defect state, it would also be possible to select the atom group belonging the defect atom only. In this example, the analysis with reduced band unfolding shows, that there are many more direct band gaps originating from the defect state.

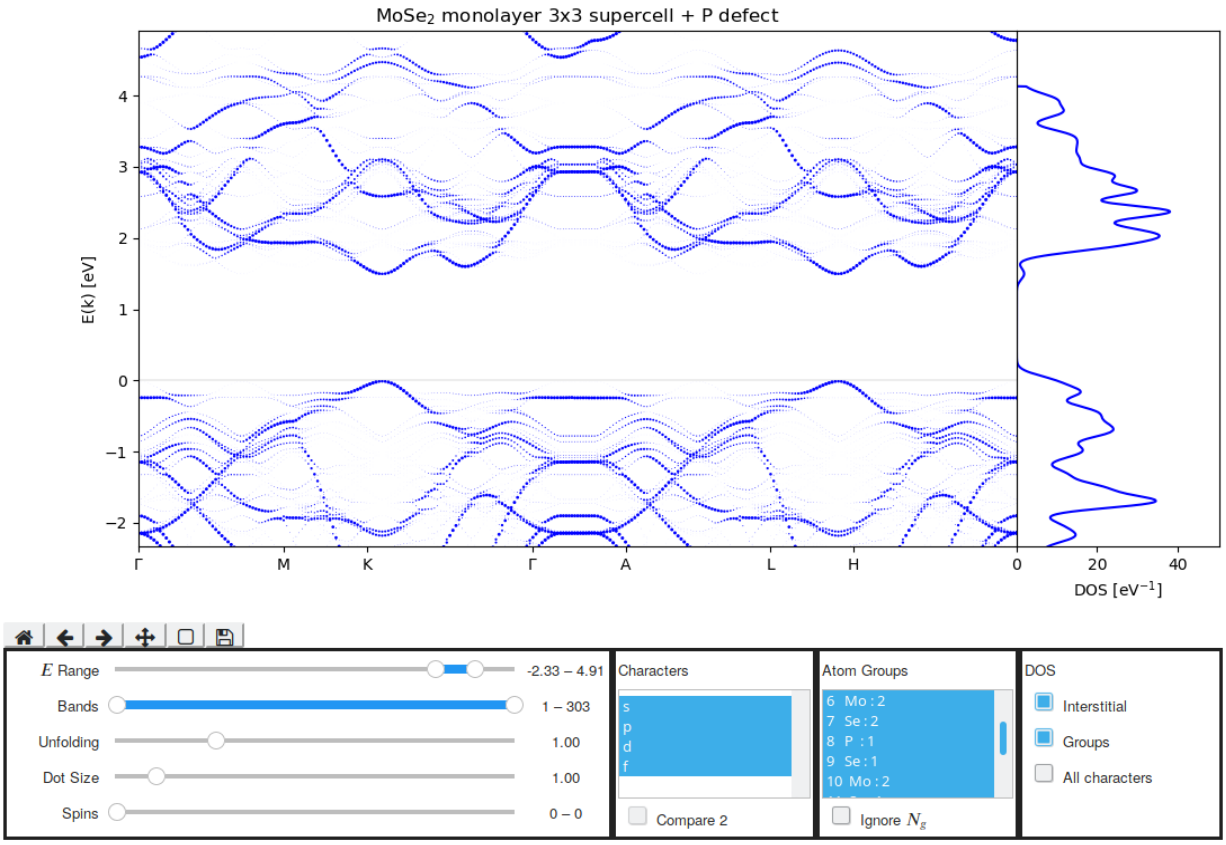


Figure 4.3: Band structure of a $MoSe_2$ monolayer using the default settings of the web frontend. Direct bandgaps are visible

Desktop Frontend: Co Crystal

Figure 4.5 shows the visualization of a band structure calculation of a three-dimensional Co crystal with the Desktop frontend. As described in the previous chapters, the possible selections in the Desktop frontend are the same as in the Web frontend and the same `matplotlib` functions are used to generate the plot, therefore the result looks very similar to the plots in chapter 4. Since Cobalt is a ferromagnet, it is interesting to investigate the spin dependence of the bands and the density of states. Therefore, for 4.5 the default settings of the GUI were modified to show both spins. As there are no defect atoms involved in this example, there is no need to perform a supercell calculation. Therefore, we do not have an unfolding weight here.

A quick look at 4.5 shows that there are several bands that intersect with the Fermi level. This indicates that in contrast to the $MoSe_2$ example, electrons can be easily excited in the conduction band without a band gap, which makes Co is a conductor.

The density of states plotted next to the band structure plot encrypts the available number of states for each spin at energy E. Since the system always minimizes its energy, only states

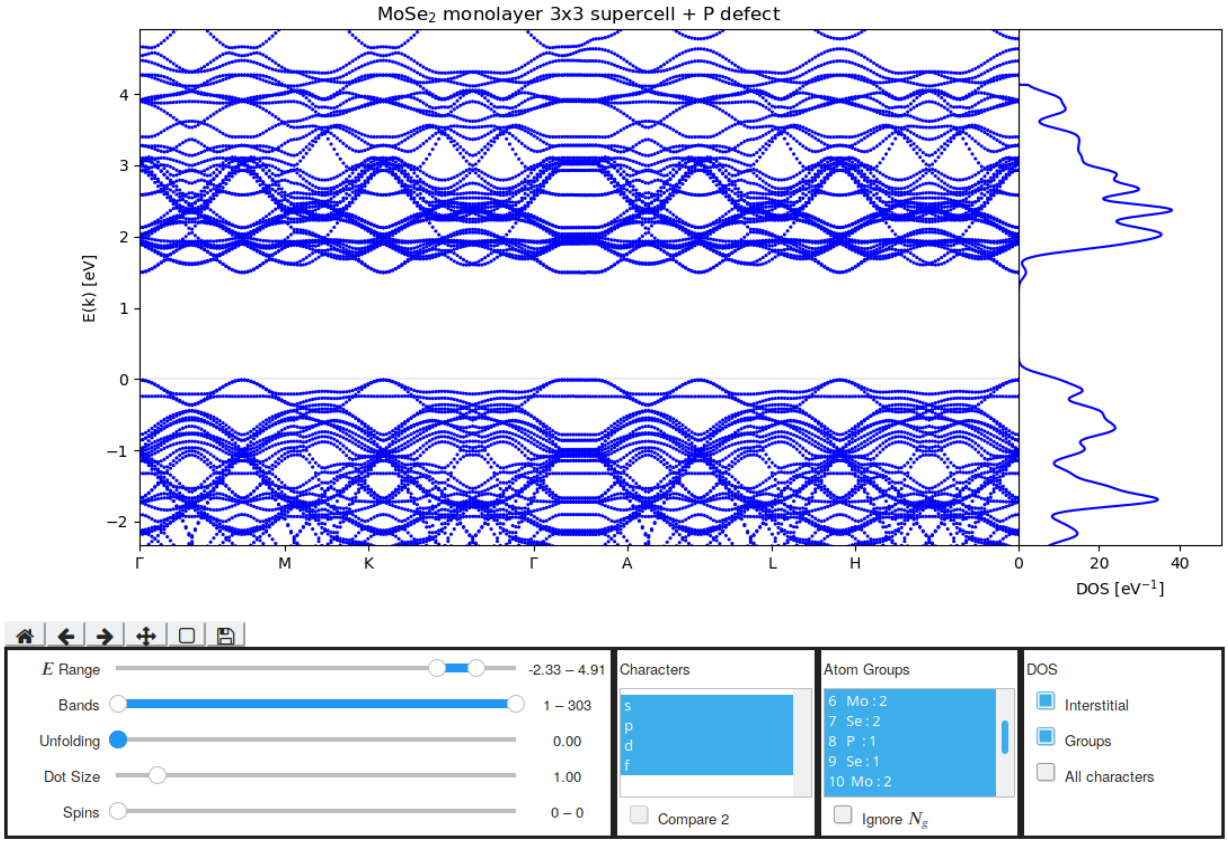


Figure 4.4: Band structure of a $\mathrm{MoSe_2}$ monolayer without unfolding weights

up to the Fermi level are occupied. Counting the number of occupied states for both spins shows an excess of blue spins over the red spins. This indicates, that the Co crystal has a net magnetic dipole moment, that contributes to the magnetic properties of Cobalt. This supports the well-known fact that Cobalt is a ferromagnetic material.

These are two properties which were deduced by observing the band-DOS plot, but depending on the material and band-DOS plot many properties can be extracted and derived.

Derived physical Quantities: Effective Mass

The kind of datasets handled in the scope of the project did not lend themselves readily to applications of automatic differentiation techniques. Nevertheless, it is possible to derive meaningful physical quantities from the band structure using numerical differentiation techniques.

The effective mass m^* represents the mass, that an electron appears to have due to the interatomic forces in the crystal. At every $\mathbf{k}_0$, where $E(\mathbf{k})$ has a local extremum, $E(\mathbf{k})$ can be expanded in a Taylor series with a vanishing first order term $E(\mathbf{k}) = E_0 + \frac{1}{2}\frac{\partial E(\mathbf{k})}{\partial \mathbf{k}} \cdot (\mathbf{k} - \mathbf{k}_0)^2 +$

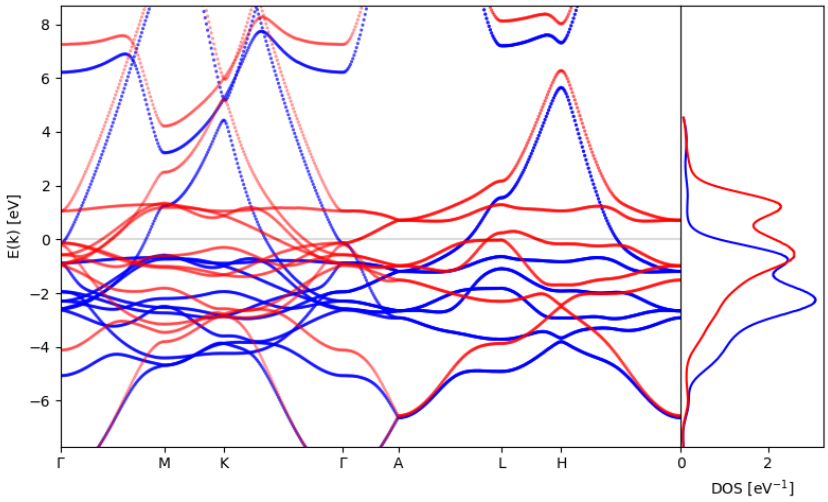


Figure 4.5: Band structure of a Co crystal in the desktop frontend. The density of states indicates a net magnetic moment in the crysal

$\mathcal{O}(\mathbf{k}-\mathbf{k}_0)^3$. Comparing this to the dispersion relation of a free electron $E(\mathbf{k})_{free} = \frac{\hbar^2\mathbf{k}^2}{2m_e}$ motivates the general definition

$$m^*_{k_i,k_j} = \hbar^2 \left(\frac{\partial^2 E(\mathbf{k})}{\partial k_i \partial k_j} \right)^{-1} \tag{4.1}$$

Since $\frac{\partial^2 E(\mathbf{k})}{\partial k_i \partial k_j}$ depends on the direction of the partial derivatives, $m^*_{k_i,k_j}$ is a tensor. Because the band structure files only contain a discretely sampled path in the Brillouin zone, only the derivatives that correspond to the direction from one high-symmetry point to the next can be computed. We are only interested in the diagonal terms of $m^*_{k_i,k_j}$.

A second potentially interesting quantity is the group velocity $v_G(\mathbf{k})$ associated to each band. The group velocity $v_G(\mathbf{k})$ at the Fermi energy $E = E_F$ is called Fermi velocity.

$$v_G(\mathbf{k})_{k_i} = \frac{1}{\hbar} \frac{\partial E(\mathbf{k})}{\partial \mathbf{k_i}} \tag{4.2}$$

Differentiation

Since the k-mesh in DFT calculations is potentially very sparse, low order finite difference schemes are not expected to work well. Alternatively, one way to exploit all data points efficiently is to use fast Fourier transform methods, which are equivalent to the derivation of a truncated Fourier series. This method is expected to be well-suited for the problem since the graph of the band structure $(k, E(|k|))$ with $k \in [-\Gamma, H, \Gamma)$ is periodic, where Γ and H are arbitrary

representatives of the high symmetry points. This periodicity in the reciprocal space is a direct consequence of the periodicity of the crystal.

In Fourier space, spatial derivatives transform into multiplications, which can easily be shown by partial integration.

$$f^{(n)}(x) = \mathcal{F}^{-1}\left((ik)^n \mathcal{F}(f(x))\right) \tag{4.3}$$

To test the FFT differentiation method, a band was selected that did not have intersections with other bands within the interval between two high symmetry points. Then a resolution study was done to investigate the impact of the number of points within the interval. The comparison between the FFT and a first-order central difference approximation of the second derivative (FD) is shown in Fig. 4.6, where different k-mesh resolutions are compared to a derivative that is almost fully converged. The comparison indicates that for small N, the error of the FFT method is significantly smaller than the error of the FD derivative. This is especially striking in the vicinity of the high symmetry points, where the error of the differentiation is required to be small in order to get good approximations for m^*, which is only meaningful close to the high symmetry points.

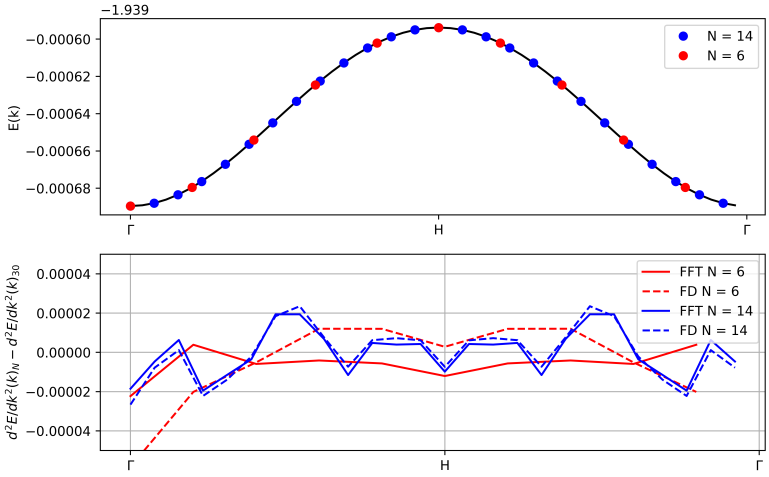


Figure 4.6: Comparison between FFT differentiation method and central finite difference method

When using more points, the difference between both approximation schemes is not significant. It is interesting to note, that the FFT method does not work anymore in the limit of extremely many points. In this case, the derivative is dominated by Gibbs oscillations. These are most likely caused by small discontinuities in $E(\mathbf{k})$, due to the different basis size at each k point in the DFT computation.

At the current state, the question remains to be answered whether the computation of m^* and v_g is useful. The number of bands, which the computation can be applied to is extremely limited. Since the bands in the data files are not labeled according to their corresponding eigenfunction but sorted by value, there are discontinuities at each point, where two bands intersect. This problem might be solved in the future.

Chapter 5

Conclusion & Outlook

During the project, we learned a lot about the workflow in electronic structure computations. This general knowledge could be put into action developing a software package that visualizes important outputs of the DFT code Fleur in a physically meaningful, intuitive and fast way. The data is read, extracted and visualized in a highly modular way such that future modifications can be implemented easily. The software is designed such that Fleur data becomes accessible to a wider range of users including non-experts since processing and visualization of the raw data is done by the software automatically. Furthermore, the software is designed to be easy accessible. Especially the desktop frontend can be installed just by copying the corresponding `.exe` file. In the future, the backend might even be integrated into AiiDA workflows and the Web frontend into AiiDA Lab, which will make it easier for anyone to access simulation results and understand the simulation much faster. Furthermore, the raw data was used to derive the effective mass m^* and Fermi velocity $v_G(\mathbf{k})$, using numerical differentiation techniques, which is helpful in analyzing materials characteristics.

During the development phase of the deliverable, several implementation and extension ideas emerged. The following list gives an indication of which milestones have yet to be reached, and where the groundwork laid in this project could lead, in no particular order of preference.

- Preprocessor submodule:
  - In the context of general workflows, preprocessor recipes could be written which serve as 'code glue' or conversion layer between different simulation programs. An example might be the conversion of a converged Fleur DFT calculation as a reference mean-field system input for Spex, another code in the Jülich FLAPW code family that uses the GW approximation [FBS10].
  - Improve the code quality by adding unit tests using a testing framework, `pytest` for instance.
  - A recipe creator has to tell the transform types which dataset dependencies to look for by entering them in a list, in the current version. This burden could be lifted via

type introspection as well.

  - The data selection method of the output type `FleurBands` could be optimized even more by replacing *all* numerical operations with `numpy` routines.

- Visualization module and frontends:
  - The 3D atoms plots are both defined in the frontends and not integrated into the visualization modules yet. Furthermore, the desktop frontend version only uses atom groups coloring, while the web frontend version only uses selected atom groups coloring.
  - A 3D visualization of the k-path in the Brillouin zone was intended but has yet to be implemented. An example of how this feature might look can be tried out in the AiiDA-based Materials Cloud tool SeeK-path).
  - The calculation of the effective mass as described in the applications chapter 4 on page 16 has not been integrated into the frontend, since the required isolation of suitable bands was found to be only possible for very simple systems. However, should the Fleur HDF output format be extended to contain unambiguous band labels, this feature could be integrated into the frontend as well.
  - The band structure, density of states, effective mass, and group velocity certainly are not the only properties that could be extracted from a Fleur simulation output. Another option for which the presented submodules preprocessing, visualization and possibly frontend could serve as a template, is the creation of small single-purpose apps to calculate physical phenomena that are not readily discernible from the simulation output, such as the refraction index of a material.
  - The fact that the web frontend is only in the experimental stage and not yet publishable as a standalone app is a drawback. It is hoped, however, that the instructions for that in the developer section of the manual A on page 27 will help to realize that milestone more easily.
  - Publishing the web frontend in AiiDA Lab still does not constitute the tool as completely open-access since it is bound to a Jupyter environment. But the deployment as a truly stand-alone web app for immediate access, like for instance the Materials Cloud Tools, the Materials Project Apps, would likely require a third frontend built on one of the popular Python Web Frameworks. As outlined in the tool selection survey [Was19], though there are tools for converting Jupyter Notebooks to standalone websites under development, none of them seemed to be production-ready yet.
  - The open issues mentioned in the developer section can be addressed to improve the user experience and the performance, yet they do not compromise the frontend usage significantly.

- PyViz Param [Ste+19] makes it possible to decouple the formal description of a particular GUI from the GUI library used. This would serve to separate interface and implementation like it is done in the preprocessor and visualization submodules.

Appendix A

Manual

The project code and documentation is hosted on the `masci-tools` repository [RBR18] under the branch `studentproject18ws`. All of the project code resides in the folders `binder` (for the web frontend demo) and `studentproject18w` (all code and documentation). The `README.md` in the latter folder serves as the manual. Therefore, the remaining part of this chapter is a TEX-ified version of that `README.md`, commit `df92930aced0dd496a29ee8cac0c67a6140dadfb`.

SiScLab 2018 Student Project **Analysis Tool for Materials Design**. Written in Python3.

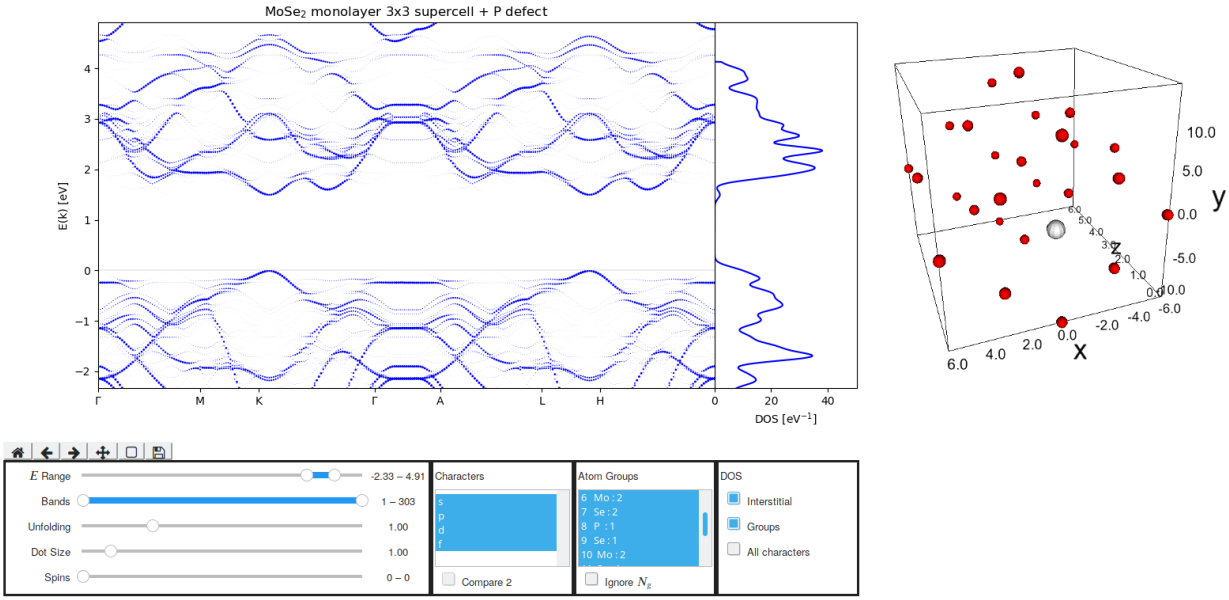
Authors: Johannes Wasmer, Christian Partmann, and Praneeth Katta.

Overview

This subfolder `studentproject18ws` is currently a largely independent side-project accompanying the main module `masci-tools`. It was created in a student project at FZJ PGI-1, and consists of three submodules:

- preprocessor: a general HDF5 reader interface, and one implementation for band structure simulation output of the DFT code Fleur
- visualization: a plotting interface, and one implementation for Fleur band structure and density of states (DOS) plots
- frontends: a desktop GUI and a web dashboard (Tk and Jupyter) for interactive Fleur band-DOS plots.

A more thorough description and example use cases can be found in the project report and presentation.



For Frontend Users

General Remarks

These remarks apply to all frontends.

Though the desktop and web frontend are functionally identical, there might be small differences in how the controls are used and how they are labeled.

File Input

The frontends currently expect band structure data in the HDF output format of Fleur. The DOS data is expected to be in the CSV output format of Fleur, one file per spin. If no DOS files are supplied, the frontend will just draw a band structure plot (band plot) and omit the adjoined DOS plot. Thus, in the following band-DOS plot stands for both kinds of plot.

Desktop Frontend

Access

A windows executable file (.exe) is made by packing all the required packages into the file. Any modern PC running on windows can run the frontend without any installation process. You don't need Python or specific packages installed.

The executable for Windows can be downloaded here. Executables for other operating systems

may be requested from the developers.

Usage

The desktop-based GUI is easy to use. Running the .exe file will open up the frontend with all packages loaded. The GUI consists of three tab windows. In the first tab window, absolute paths to the input data files must be entered in this order: HDF and (optional) DOS file for spin '0' and '1'. Tab 2 shows the band-DOS plot, and Tab 3 shows the 3D atoms plot. After loading the files, the controls must be initialized. Finally, clicking the 'Update' button produces the plots.

Controls for all three plots:

- **Update, SaveButton**: Redraw the plots with the new selection / save the the plot as a PDF.
- **Atom Groups**: Redraw only for the selected symmetry groups.

Controls for the band and DOS plots:

- **Ymin, Ymax**: Limit the energy range.
- **BandMin, BandMax**: Limit the band range.
- **Character**: Likewise for the characters (orbitals) 'S','P','D','F'.
- **Spin**: Draw bands and DOS for one or both spins.

Controls for the band plot only:

- **Marker Size**: Set the marker size of the dots (eigenenergies).
- **Exponential weight**: The unfolding exponent for supercell calculations (see report). Value 0.0 means no unfolding weight. If the calculation is done with a unit cell, this control has no effect.
- **Compare 2 Characters**: Show the respective contribution of the two selected characters to each eigenergy using a sequential (2) colormap. Disabled if other than two characters are selected.
- **Ignore Atom Group**: Set the contribution of each group to be equal instead of to the number of atoms per group. Helpful to investigate defect states.

Controls for the DOS plot only:

- **Select groups**: Include the selected atom groups in the DOS.
- **Interstitial**: Include the contribution from interstitial region in the DOS.
- **All characters**: Include all characters in the DOS regardless of character selection. In the DOS CSV file, different input data is used (a summed column).

Troubleshooting

If the band plot is not visible:

- Click update two to three times.
- Check if the three input files (if any) are belonging to the same Fleur calculation and selected appropriately.
- Check if at least one Atom Group, one Character, one Spin is selected.
- Check if Ymin is less than Ymax and similarly BandMin is less than BandMax such that software is able to plot.

If the DOS plot is not visible:

- Make sure either Select Groups or Interstitial is selected.

If the problem persists, try restarting the GUI. If that fails, please open an issue to contact the developers.

Web Frontend

Access

The web frontend is a Jupyter Dashboard. It is in experimental state (no fileupload yet). You can try it out here on Binder. You can also run it locally (see developer section). If you have an AiiDA Lab account: the dashboard is planned to be published as an app there.

Usage

Using the Dashboard should be self-explanatory to the domain user. Some tips:

- If the plot window is not on startup or gets stuck, reload/rerun once.
- Plot updates are instantaneous.
- Empty selections are impossible.
- Only shows controls for data that is present in the input.
- Multi-selection boxes: use ctrl or shift to select multiple items.
- Slider values can also be typed into the adjoining text box.
- Try out the zoom and pan tools below the plot, they're useful.

For Developers

Installation

Clone this repo (branch). Then create a virtual environment for the project.

With conda (recommended): - Install Anaconda (3 recommended) - Install the environment `masci-stupro` with the necessary and recommended dependencies:

```
conda create -f environment.yml
source activate masci-stupro
```

With virtualenv (untested):

```
virtualenv masci-stupro
source masci-stupro/bin/activate
pip install -r requirements_pip.txt # install requirements
```

Programmatic use

Though `masci-tools` is (planned to be) availabe via PyPI, there is currently no plan to integrate `studentproject18ws`. If you want to use it in your code, use it in an IDE, or append the path to your `sys.path`:

```
import sys
if path_repo not in sys.path:
    sys.path.append(path_repo)

# now import works
from studentproject18w.hdf.reader import Reader
# ...
```

In this example, a Fleur HDF file is preprocessed using the Recipe `FleurBands`. The resulting output `data` with the extracted and transformed HDF datasets and attached load methods (Extract-Transform-Load) is then passed to a plotter, alongside some DOS CSV files for a bandstructure plot using `matplotlib` as backend library.

```
from studentproject18w.hdf.reader import Reader
from studentproject18w.hdf.recipes import Recipes
from studentproject18w.plot.matplot import BandDOSPlot
import matplotlib.pyplot as plt

data = None
reader = Reader(filepath=filepath_hdf)
```

```
with reader as h5file:
    data = reader.read(recipe=Recipes.FleurBands)
    #
    # Note:
    # Inside the with statement (context manager),
    # all data attributes that are type h5py Dataset are available
    # (on-disk access). When the statement is left, the HDF5 file
    # gets closed and the datasets are closed.
    #
    # Use data outside the with-statement (in-memory access: all
    # HDF5 datasets converted to numpy ndarrays):
    data.move_datasets_to_memory()

plotter = BandDOSPlot(plt, data, filepaths_dos)
(fig, ax_bands, ax_dos) = plotter.setup_figure()

data_selection = some_selection_process()
plotter.plot_bandDOS(*data_selection)
plt.show()
```

Try Out the Web Frontend Locally

The demo notebook with the dashboard is `studentproject18w/frontend/jupyter/demo/demo.ipynb`.

If Using Jupyter Notebook

On Windows, omit keyword `source`.

```
source activate masci-stupro
cd mypath/masci-tools/studentproject18ws/
jupyter-notebook .
# if Home is not set to this dir, try this instead:
# /home/you/anaconda3/envs/myenv/bin/python /home/you/anaconda3/envs/myenv/bin/
    ↪jupyter-notebook .
```

If using Jupyter Lab

Additional installation step needed:

```
source activate masci-stupro
```

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager jupyter-matplotlib ipyvolume
cd mypath/masci-tools/studentproject18ws/
jupyter-lab
```

Frontend Deployment

Desktop Frontend

To create executables for different operating systems, use PyInstaller. The target file is `frontend/tkinter/gui.py`.

```
source activate masci-stupro
conda install -c conda-forge pyinstaller
# if not using conda: pip install pyInstaller
cd mypath/masci-tools/studentproject18ws/
pyinstaller --onefile frontend/tkinter/gui.py
```

Web Frontend

The web frontend is currently a single Jupyter notebook. In order to publish it as a usable standalone app, additional work has to be done.

- Create `frontend/jupyter/Dashboard.py` and put code of demo_backend.ipynb notebook inside it. This will become the widget. Use aiidalab-widgets-base > StructureUploadWidget as a template. Create `frontend/jupyter/Dashboard.ipynb` notebook. This will become the app. Use StructureUploadWidget demo notebook as a template.
- Add fileupload buttons (for HDF, DOS) to widget (again, like in StructureUploadWidget. See binder_fileupload_test.ipynb notebook for a demo that works with binder.)
- Now the web frontend should work on Binder.
- For publishing the app on AiiDA Lab, it has to be registered in the aiidalab-registry.
  - Create a skeleton using the aiidalab-app-cutter.
  - The project code is in Python3, but aiidalab requires Python2. So the code has to first be backported by hand using the `future` package. If this takes too long, maybe try the tool 3to2.
  - Use the simplest app in the registry, aiidalab-units as a template. Adapt code.
  - Try it out first in the Quantum Mobile Virtual Machine, which has aiidalab installed and configured. Else try it in a virtual environment with aiidalab installed from PyPI.
  - Register the app.

Note: other publishing options besides Binder and AiiDALab are listed here. For instance, Google Colaboratory is a free notebook hosting service that allows environment creation and file upload.

Extending the code

Use Case: HDF format that includes DOS data

The Fleur output HDF format is expected to change and incorporate more data. In turn, this project's code has to be extended as well. The procedure is outlined for a an example use case: the incorporation of DOS data into the band structure HDF (thus eliminating the need for separate DOS CSV files). The instructions show how to extend the preprocessor, the visualization and frontend submodules to that scenario.

- Add a new output type to `hdf/output_types`, say `FleurBandDOS`. Let it inherit from output type `FleurBands`. If you want an output type just for the DOS as well, add a type `FleurDOS` and let `FleurBandDOS` inherit it.
- Add a new recipe to `hdf/recipes` e.g. `FleurBandDOS`. Copy unchanged things from recipe `FleurBands`.
- If needed, add new transforms to `hdf/input_transforms`. Adhere to the transform function standard there. If there are mutual dependencies, add them to the list in the top of the file.
- Add a DOS data selection method to the output type `FleurBands`. The `DOSPlot` in `plot/base` types will need those to plot the DOS plot. Simply adapt from the function in `dos/reader` for the DOS CSV files, adopt the identical signature.
- In the `DOSPlot` types in submodule `plot`, add a switch to the constructor that can distinguish the three cases (bands, bands+CSV DOS, bands+HDF DOS). Use the switch in the `plotDOS` methods, and for the case bands+HDF DOS, call your new `FleurBandDOS` function.

Extending the Visualization (Plots)

- In addition to the inheritance scheme based on Python `AbstractBaseClass` (ABC) detailed in the report, the `Plot` types in `plot` have an additional facility that helps to keep the appearance of different frontends synchronized: each type has an attribute `icdv` of type `InteractiveControlDisplayValues`. This is an ABC with the same inheritance as the application Plot types. For every plot control argument that an application type's Plot type exposes in it's methods' signatures, this attribute describes the parameters of the accompanying control widget in the frontend (text label, default values, value ranges, and so on). In the current code, only the web frontend uses this

facility, so the labels in the desktop frontend differ slightly.

- It is worth pointing out that unlike other languages, Python does not enforce implemented abstract methods to have the same method signature. However, when a new implementation for a different plotting library/backend is added, it is recommended to adopt the `abstractmethod` signature. That way, changing the backend in a use case only requires to change the import.

Open Issues

- Running the frontends in a debugger or with a counter reveals: on a plot selection change or update, the plot seems to be redrawn not once but several times. The cause could not be found so far.
- In the desktop frontend, the update button has to be clicked several times.
- In the web frontend, on startup, the plot is only visible after two loads/cell runs.

Bibliography

[Aut18] PyViz Authors. *PyViz FAQ*. http://pyviz.org/FAQ.html. PyViz Team. 2018.

[Bed18] James Bednar. *Python Data Visualization 2018*. Blog. https://www.anaconda.com/python-data-visualization-2018-why-so-many-libraries/ and https://www.anaconda.com/python-data-visualization-2018-moving-toward-convergence/ and https://www.anaconda.com/python-data-visualization-2018-where-do-we-go-from-here/. 2018.

[Blü+18] Stefan Blügel et al. *Fleur: full potential linearized augmented planewave code*. https://iffgit.fz-juelich.de/fleur/fleur/. 2018.

[Dou+08] John R Douceur et al. "Leveraging Legacy Code to Deploy Desktop Applications on the Web." In: *OSDI*. Vol. 8. 2008, pp. 339–354.

[FBS10] Christoph Friedrich, Stefan Blügel, and Arno Schindlmayr. "Efficient implementation of the G W approximation within the all-electron FLAPW method". In: *Physical Review B* 81.12 (2010), p. 125102. URL: https://spex.readthedocs.io/en/master/.

[HK64] Pierre Hohenberg and Walter Kohn. "Inhomogeneous electron gas". In: *Physical review* 136.3B (1964), B864.

[Hof87] Roald Hoffmann. "How chemistry and physics meet in the solid state". In: *Angewandte Chemie International Edition in English* 26.9 (1987), pp. 846–878.

[Ind08] Inductiveload. *A diagram of the first Brillouin zone of a face-centred cubic (FCC) lattice, with pointsof high symmetry marked.* File: Brillouin Zone (1st, FCC).svg. 2008. URL: https://commons.wikimedia.org/wiki/File:Brillouin_Zone_(1st,_FCC).svg.

[Koh99] W. Kohn. "Nobel Lecture: Electronic structure of matter—wave functions and density functionals". In: *Rev. Mod. Phys.* 71 (5 Oct. 1999), pp. 1253–1266. DOI: 10.1103/RevModPhys.71.1253. URL: https://link.aps.org/doi/10.1103/RevModPhys.71.1253.

[Kor11] Sandeep Koranne. "Hierarchical Data Format 5 : HDF5". In: *Handbook of Open Source Tools*. Boston, MA: Springer US, 2011, pp. 191–200. ISBN: 978-1-4419-7719-9. DOI: 10.1007/978-1-4419-7719-9_10. URL: https://doi.org/10.1007/978-1-4419-7719-9_10.

[KS65] Walter Kohn and Lu Jeu Sham. "Self-consistent equations including exchange and correlation effects". In: *Physical review* 140.4A (1965), A1133.

[Meu+14] Hans Werner Meuer et al. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. 1st. Chapman & Hall/CRC, 2014. ISBN: 143981595X.

[Piz+16] Giovanni Pizzi et al. "AiiDA: automated interactive infrastructure and database for computational science". In: *Computational Materials Science* 111 (2016), pp. 218–230. ISSN: 0927-0256. DOI: https://doi.org/10.1016/j.commatsci.2015.09.013. URL: http://www.sciencedirect.com/science/article/pii/S0927025615005820.

[RBR18] Philipp Rüßmann, Jens Bröder, and Stefan Rost. *masci-tools*. https://github.com/JuDFTteam/masci-tools. 2018.

[Sch15] Johannes Schneider. *Elementarzelle einer kubisch flächenzentrierten Kristallstruktur*. File: Elementarzelle einer kubisch flächenzentrierten Kristallstruktur.png. 2015. URL: https://commons.wikimedia.org/wiki/File:Elementarzelle_einer_kubisch_fl%C3%A4chenzentrierten_Kristallstruktur.png.

[Ste+19] Jean-Luc Stevens et al. *PyViz Param*. https://github.com/pyviz/param. 2019.

[VR17] Jake VanderPlas and Nicolas Rougier. *Python Visualization Landscape*. https://github.com/rougier/python-visualization-landscape. 2017.

[Was19] Johannes Wasmer. *Software Development Notes*. https://github.com/Irratzo/notes. 2019.