AiiDA Wannier90 Tutorial (2020)

Mar 23, 2020

TUTORIAL MATERIALS

This document contains the hands-on for the AiiDA+Wannier90 tutorial, held in March 2020 as an online "virtual event".

CHAPTER

ONE

2020 WANNIER TUTORIAL "VIRTUAL EDITION"

Related resources	
Virtual Machine	Quantum Mobile 20.03.1
python packages	aiida-core 1.1.0, aiida-quantumespresso 3.0.0a6 , aiida-wannier90 2.0.0, aiida-wannier90-workflows 1.0.1
codes	Quantum Espresso 6.5, Wannier90 3.1.0

Note: The credentials of the Quantum Mobile VM are the following: username: `max`, password: `moritz`. Also, when you start the VM, to make it more performant you might want to set 2 CPUs, and 2GB of RAM or more. If the machine becomes slow after a few hours of use, in VirtualBox 6 you can change the settings of the graphics controller to VMSVGA.

These are the hands-on materials from the 1-day AiiDA tutorial, part of the Wannier90 v3.0: new features and applications, that should have been held on March 25-27, 2020 in Oxford (UK) but was converted to a "Virtual Edition" (as discussed on the online page).

1.1 AiiDA-Wannier90 hands-on

1.1.1 Demo

Getting set up

Download the correct Quantum Mobile version listed *in the homepage of this event*.

Note: The credentials of the Quantum Mobile VM that you can download from the main page of this tutorial material are the following: username: `max`, password: `moritz`.

You can find some troubleshooting, in case you have problems, on the Quantum Mobile FAQ page.

Start jupyter

Once connected to your virtual machine, to use AiiDA open a terminal and type

```
workon aiida
```

This will enable the virtual environment in which AiiDA is installed, allowing you to use AiiDA. Now, to start a jupyter notebook (you might need it later in the tutorial) type in the same shell:

```
jupyter notebook
```

This will run a server with a web application called `jupyter`, which is used to create interactive python notebooks. This will also open the default browser (in the VM) and allow you to use jupyter. Keep the browser open, so you can use jupyter later when needed.

Keep also the terminal open (just minimise the window). If you need to use the terminal, just open another one (right click on the terminal icon on the left bar, and choose "New terminal").

Additional useful notes

Read through the next sections if you are interested in knowing how to download files into the virtual machine, how to troubleshoot typical problems, or how to get additional help.

Downloading files

Throughout this tutorial, you will encounter links to download python scripts, jupyter notebooks and more. These files should be downloaded to the environment/working directory you use to run the tutorial. In particular, when running the tutorial on a linux virtual machine, copy the link address and download the files to the machine using the `wget` utility on the terminal:

```
wget '<LINK>'
```

where you replace `<LINK>` with the actual HTTPS link that you copied from the tutorial text in your browser. This will download that file in your current directory.

Troubleshooting

- If you get errors `ImportError: No module named aiida` or `No command 'verdi' found`, double check that you have loaded the virtual environment with `workon aiida` before launching `python`, `ipython` or the `jupyter` notebook server.

Getting help

There are a number of helpful resources available to you for getting more information about AiiDA. Please consider:

- consulting the extensive AiiDA documentation
- opening a new issue on the tutorial issue tracker
- asking your neighbor
- asking a tutor.

A first taste of AiiDA

Let's start with a quick demo of how AiiDA can make your life easier as a computational scientist.

We'll be using the `verdi` command-line interface, which lets you manage your AiiDA installation, inspect the contents of your database, control running calculations and more.

As the first thing, open a terminal and type `workon aiida` to enter the "virtual environment" where AiiDA is installed. You will know that you are in the virtual environment because each new line will start with `(aiida)`, e.g.:

```
(aiida) max@qmobile:~$
```

Note that you will need to retype `workon aiida` every time you open a new terminal.

Here are some first tasks for you:

- The `verdi` command supports **tab-completion**: In the terminal, type `verdi`, followed by a space and press the 'Tab' key twice to show a list of all the available sub commands.
- For help on `verdi` or any of its subcommands, simply append the `--help/-h` flag:

  ```
  verdi -h
  ```

Note: This tutorial is a short crash course into AiiDA, focusing on Wannier90 as the code that AiiDA will run (via the aiida-wannier90 plugin).

Many more codes are supported by AiiDA (see full updated list of plugins on the AiiDA plugin registry). If you want to see a similar first-taste tutorial, but focused on Quantum ESPRESSO instead (using the aiida-quantumespresso plugin), you can check the "first-taste" page of the tutorial held in Ljubiana (2019).

Importing a crystal structure from a file

We will use a gallium arsenide (GaAs) primitive cell with 2 atoms in this first part of the tutorial.

We provide the crystal structure, in XSF format, here: `GaAs.xsf`.

You can download the file in a folder in the virtual machine, and then import it into AiiDA (via the ASE library) using the following `verdi` command, run from a bash shell:

```
verdi data structure import ase GaAs.xsf
```

Each piece of data in AiiDA gets a PK number (a "primary key") that identifies it in your database. This is printed out on the screen by the `verdi data structure import` command. Mark it down, as we are going to use it in the next commands.

Note: In the next commands, replace the string `<PK>` with the appropriate PK number.

Let us first inspect the node you just created:

```
verdi node show <PK>
```

You will get in output some information on the node, including its type (`StructureData`, the AiiDA data type for storing crystal structures), a label and a description (empty for now, can be changed), a creation time (`ctime`) and a last modification time (`mtime`), the PK of the node and its UUID (universally unique identifier).

Note: When should I use the PK and when should I use the UUID?

A **PK** is a short integer identifying the node and therefore easy to remember. However, the same PK number (e.g., PK=10) might appear in two different databases referring to two completely different pieces of data.

A **UUID** is a hexadecimal string that might look like this:

```
d11a4829-3e19-4978-bfcf-c28ddeb0891e
```

A UUID has instead the nice feature to be globally unique: even if you export your data and a colleague imports it, the UUIDs will remain the same (while the PKs will typically be different).

Therefore, use the UUID to keep a long-term reference to a node. Feel free to use the PK for quick, everyday use (e.g. to inspect a node).

Note: All AiiDA commands accepting a PK can also accept a UUID. Check this by trying the command above, this time with `verdi node show <UUID>`.

Note the following:

- AiiDA does not require the full UUID, but just the first part of it, as long as only one node starts with the string you provide. E.g., in the example above, you could also say `verdi node show d11a4829-3e19`. Most probably, instead, `verdi node show d1` will return an error, since you might have more than one node starting with the string `d1`.
- By default, if you pass a valid integer, AiiDA will assume it is a PK; if at least one of the characters is not a digit, then AiiDA will assume it is (the first part of) a UUID.
- How to solve the issue, then, when the first part of the UUID is composed only by digits (e.g. in `2495301c-dd00-42d6-92e4-1a8c171bbb4a`)? As described above, using `verdi node show 24953` would look for a node with `PK=24953`. As a solution, just add a dash, e.g. `verdi node show 24953-` so that AiiDA will consider this as the beginning of the UUID.

  Note that you can put the dash in any part of the string, and you don't need to respect the typical UUID pattern with 8-4-4-4-12 characters per section: AiiDA will anyway first strip all dashes, and then put them back in the right place, so e.g. `verdi node show 24-95-3` will give you the same result as `verdi node show 24953-`.

- Try to use again `verdi node show` on the `StructureData` node above, just with the first part of the UUID (that you got from the first call to `verdi node show` above).

Running a job calculation

Introduction and importing existing simulations

In AiiDA, one very important type of calculation is called "calculation job" (whose logic is implemented in a `CalcJob` python class, and that is stored upon execution using a `CalcJobNode` python class). These calculations represent the execution of an external code (e.g. Quantum ESPRESSO, Wannier90, ...), very often on a different computer than the one where AiiDA is installed. The execution is automatically tracked by AiiDA (input creation, submission, waiting for the job scheduler, file retrieval and parsing) and its inputs and outputs are connected to the `CalcJobNode` (representing the execution) via INPUT and CREATE links.

In the following, we want to launch a `CalcJob` running a Wannier90 calculation. Typically, before running a Wannier90 calculation, you need to obtain the `.amn`, `.mmn`, ... files from the interface to a first-principles code. In order

to keep this tutorial focused on Wannier90, we have already run that part with AiiDA (using Quantum ESPRESSO) and we will just import it into your database.

To achieve so, download this AiiDA export file: `example-gaas-wannier.aiida` and, once you have downloaded it in the current folder, run the following command in your bash shell:

```
verdi import example-gaas-wannier.aiida
```

This will import, in particular, the node with UUID `71155a0b-6cb9-4712-a043-dc4798ccfaaf`, that contains the `.amn`, `.mmn`, ... files created by the `pw2wannier90.x` code of Quantum ESPRESSO. Its provenance looks like the following figure (with some output nodes of the calculations not shown for clarity):

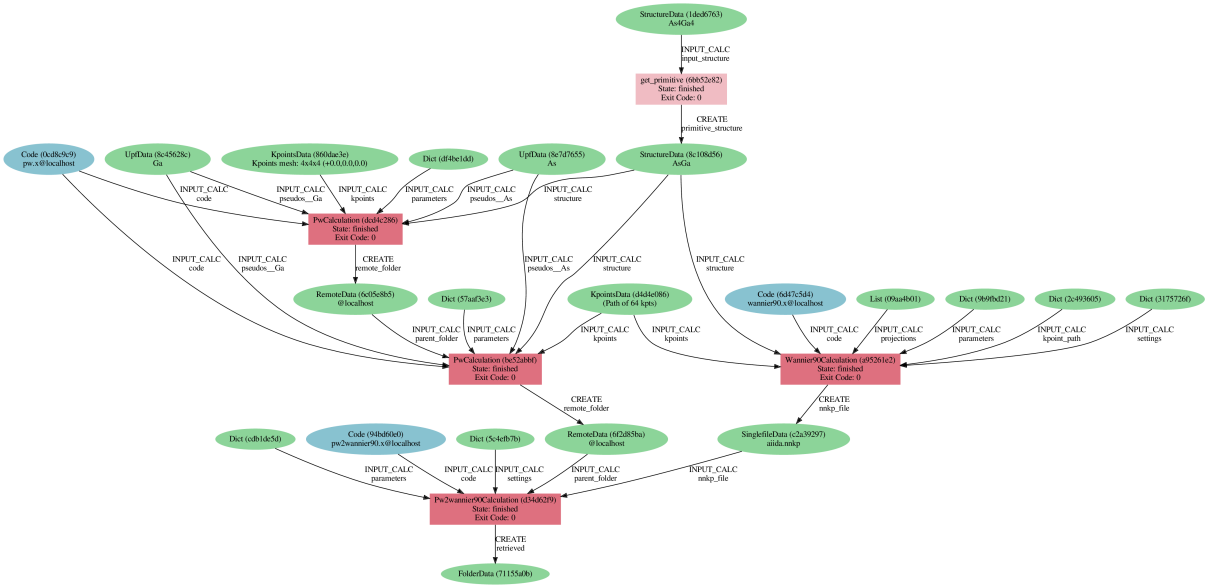


Fig. 1.1: Provenance graph for the `FolderData` node (`71155a0b`) that we have already run. At the top, we see the execution of a `get_primitive()` calc function (described later in *the appendix*) to obtain a primitive cell from a conventional cell. The darker red rectangles represent the various calc jobs that we have run for you, in particular:

- the Quantum ESPRESSO SCF step (UUID `dcd4c286`)
- the Quantum ESPRESSO NSCF step (UUID `be52abbf`)
- the Wannier90 preprocess (`-pp`) step (UUID `a95261e2`)
- the Quantum ESPRESSO pw2wannier90 step (UUID `d34d62f9`).

Exercise: To check that the import worked correctly, use `verdi node show` on the UUID of the `FolderData` mentioned above to check that you indeed have the node in your database.

Exercise: `FolderData` is a type of node that stores an arbitrary set of files and folders in AiiDA. Check the list of files included in it with the command `verdi node repo ls 71155a0b`, and check the content of a given file (e.g. the `aiida.mmn` file) with `verdi node repo cat 71155a0b aiida.mmn`.

Exercise: In the figure above, verify that all calculations are connected between them via provenance links (through some data node). Try to understand how, e.g., the NSCF calculation "restarts" from the SCF via a `RemoteData` node (that represents a reference to the folder in the computational cluster where the SCF calculation was run), or how the pw2wannier90 step uses as input both the `RemoteData` node of the NSCF and the `.nnkp` file generated by `wannier90.x -pp`.

Running Wannier90 with AiiDA

The following python script sets up all inputs to run the Wannier90 code. We will discuss relevant parts below; we will first launch it so that we can analyze its output while it runs.

```
#!/usr/bin/env runaiida
from aiida.engine import submit
from aiida.orm import load_node, Code, Dict

code = Code.get_from_string("<CODE LABEL>")  # REPLACE <CODE LABEL>

# Fill in here the PK of the *primitive* structure that you obtained
# in the tutorial. If you want to reuse the crystal structure that
# was already imported, you can use instead the following UUID: '8c108d56-aca6-43f6-
↪baa6-94f7d1d9887d'
structure = load_node(<PK>)
# Here, we are reusing the same k-points used in the NSCF step.
# Exercise: load this node in the `verdi shell`, and then use
# `kpoints.get_kpoints()` to check the full list of kpoints.
kpoints = load_node('d4d4e086-af7a-46c7-b7a0-9c5c2c9dfc7b')

## Main Wannier run
do_preprocess = False
## This is the node that we imported, discussed in the tutorial
parent_folder = load_node('71155a0b-6cb9-4712-a043-dc4798ccfaaf')

## Note: if you wanted to run a pre-process step (`Wannier90.x -pp`),
## than you would replace the two lines above with the following one:
#do_preprocess = True



parameters = Dict(
    dict={
        'bands_plot': True,
        'num_iter': 300,
        'guiding_centres': True,
        'num_wann': 4,
        'mp_grid': [4,4,4],
        'exclude_bands': [1, 2, 3, 4, 5]
    }
)

kpoint_path = Dict(
    dict={
        'point_coords': {
            'GAMMA': [0.0, 0.0, 0.0],
            'L': [0.5, 0.5, 0.5],
            'X': [0.5, 0.0, 0.5]
        },
        'path': [('L', 'GAMMA'), ('GAMMA', 'X')]
    }
)

projections = List()
projections.extend(['As:s','As:p'])
```

(continues on next page)

(continued from previous page)

```
# Settings node, with additional configuration
settings_dict = {}
if do_preprocess:
    settings_dict.update(
        {'postproc_setup': True}
    )

# Prepare the builder to launch the calculation
builder = code.get_builder()
builder.metadata.options.max_wallclock_seconds = 30 * 60  # 30 min
builder.metadata.options.resources = {"num_machines": 1}

builder.structure = structure
builder.projections = projections
builder.parameters = parameters
builder.kpoints = kpoints
builder.kpoint_path = kpoint_path
if not do_preprocess:
    builder.local_input_folder = parent_folder
builder.settings = Dict(dict=settings_dict)

# Run the calculation and get both the results and the node
calcjobnode = submit(builder)

print("CalcJobNode: {}".format(calcjobnode))
print("Use `verdi process list` or `verdi process show {}` to check the progress".
↪format(calcjobnode.pk))
```

Download the `demo_wannier_calcjob.py` script to your working directory. It contains a few placeholders for you to fill in:

1. the VM already has a number of codes preconfigured. Use `verdi code list` to find the label for the Wannier90 code and use it in the script. Note: *do not use the imported code*, as the computer on which it runs will not be configured by default.
2. replace the PK of the structure with the one you obtained earlier (*important*: use the PK of the *primitive* structure).

Then submit the calculation using:

```
verdi run demo_wannier_calcjob.py
```

From this point onwards, the AiiDA daemon will take care of your calculation: creating the necessary input files, running the calculation, and parsing its results.

In order to be able to do this, the AiiDA daemon must be running: to check this, you can run the command:

```
verdi daemon status
```

and, if the daemon is not running, you can start it with

```
verdi daemon start
```

It should take less than one minute to complete.

Analyzing the outputs of a calculation

Let's have a look at how your calculation is doing:

```
verdi process list  # shows only running processes
verdi process list --all  # shows all processes
```

Again, your calculation will get a PK, which you can use to get more information on it:

```
verdi process show <PK>
```

As you can see, AiiDA has tracked all the inputs provided to the calculation, allowing you (or anyone else) to reproduce it later on. AiiDA's record of a calculation is best displayed in the form of a provenance graph:

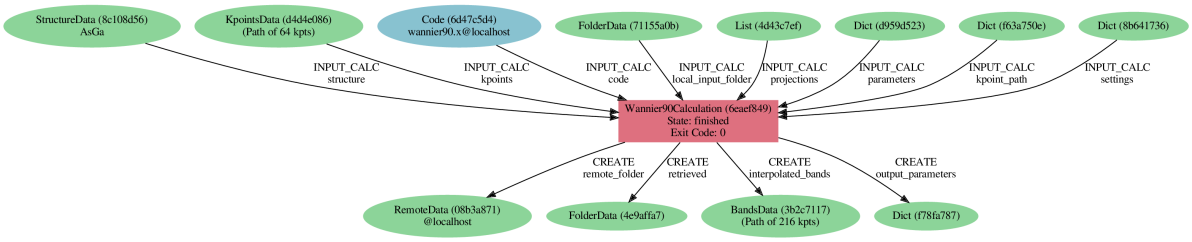


Fig. 1.2: Provenance graph for a single Wannier90 calculation.

You can generate such a provenance graph for any calculation or data in AiiDA by running:

```
verdi node graph generate <PK>
```

Try to reproduce the figure using the PK of your calculation (note that in our figure, we have used the `--ancestor-depth=1` option of `verdi node graph generate` to only show direct inputs; if you don't, you will see also the full provenance of the data, similar to the previous figure shown earlier).

You might wonder what happened under the hood, e.g. where to find the actual input and output files of the calculation. You will learn more about this in some of the advanced tutorials on AiiDA – for now, here are a few useful commands:

```
verdi calcjob inputcat <PK>  # shows the input file of the calculation
verdi calcjob outputcat <PK>  # shows the output file of the calculation
verdi calcjob res <PK>  # shows the parsed output
```

Exercise: Here are a few questions you can now answer using these commands (try to check both the raw output and the parsed

- What are the values of the various components of the spread Ω_I, Ω_D, Ω_{OD}?
- How many Wannier functions have been computed?
- Was there any warning?

Understanding the launcher script

Above, we have just provided you a script that submits a calculation. Let us now check in detail some relevant sections, and how the input information got converted into the Wannier90 raw input file.

Choosing a structure

In this simple example, we load a structure already stored in the AiiDA database (it is the one we imported from a file earlier on). We do this using the `load_node` command (that accepts either an integer PK or a string UUID).

Choosing the k-points

Also for `kpoints`, we are loading these from file. The reason is that we already executed the Quantum ESPRESSO NSCF run, and we need to be sure that the order of the k-points is the same (exercise: check in Fig. 1.1 that the `KpointsData` node that we are using is indeed the input `KpointsData` of the NSCF calculation). Otherwise, one can create an AiiDA `KpointsData` and use the `set_kpoints` method to pass a $N \times 3$ numpy array, where N is the number of k-points.

Setting up input parameters

One very important input node is the `parameters` node, containing the input flags for the code.

```
parameters = Dict(
   dict={
      'bands_plot': True,
      ...
   })
```

In particular, `parameters` is a `Dict` node, i.e., an AiiDA node containing a dictionary of key-value pairs (stored in the AiiDA database and that can be easily queried - we will not see how to run queries in this tutorial, but you can check the AiiDA documentation on querying or the full tutorial for more information on this).

Exercise: Use `verdi calcjob inputcat <PK>` (using the `PK` of the calc job) and check how the information in the `parameters` has been converted into the Wannier90 input file.

Setting up the projections

In `aiida-wannier90`, there are two ways to set up `projections`. Here, we are using the simplest approach (i.e., a list of strings, that are simply inserted in the corresponding section of the raw input file). Similarly to a `Dict` node, these strings are wrapped in a `List` AiiDA node, so that it gets stored and are queryable.

We will see a second (more easily queryable) way later on during this tutorial.

Setting up an (optional) list of k-points for the band plot

The path to compute an (interpolated) band structure needs to be specified as a `Dict` node, contianing two keys: `point_coords` (a dictionary with high-symmetry point labels and coordinates), and a `path` (a list of pairs of labels, to indicate band-path segments):

```
kpoint_path = Dict(
   dict={
      'point_coords': {
            'GAMMA': [0.0, 0.0, 0.0],
            'L': [0.5, 0.5, 0.5],
            'X': [0.5, 0.0, 0.5]
      },
```

(continues on next page)

(continued from previous page)

```
        'path': [('L', 'GAMMA'), ('GAMMA', 'X')]
    }
)
```

Exercise: Check how this has been converted into the raw input file.

Setting up the (optional) calculation settings

While often you don't need to specify additional control parameters, there are cases in which you want to specify additional options to tune the way AiiDA will run the code.

You can find the full documentation of accepted keys for the `settings` in the aiida-wannier90 documentation (e.g. to specify random projections, or to retrieve/not retrieve specific files at the end of the run).

One important option that you will use very often is the flag `postproc_setup`: if set to `True`, it will run a Wannier90 post-processing run (i.e., `wannier90.x -pp`).

In our case, `do_preprocess` is `False` so this is not set, but you would need to run it in a full calculation including a DFT code.

```
# Settings node, with additional configuration
settings_dict = {}
if do_preprocess:
    settings_dict.update(
        {'postproc_setup': True}
    )
```

Setting up all inputs in a builder

Once you have created the nodes or loaded them from the database, you need to prepare a calculation, connect all inputs and then run (or submit it).

This is done creating a new calculation `builder` from a code: `builder = code.get_builder()`. Then, all inputs are attached to the builder as follows:

```
builder.structure = structure
builder.projections = projections
builder.parameters = parameters
...
```

Finally, you can:

- `run` the calculation (this stops the interpreter at that line, until the simulation is done, and then the execution of the python script continues):

  ```
  from aiida.engine import run
  results = run(builder)
  ```

  At the end, you will get a dictionary of `results`, one for every output node.

- If you want to run the calculation, but also get a reference to the `CalcJobNode` that represents the execution, you can use instead:

  ```
  from aiida.engine import run_get_node
  results, calcjobnode = run_get_node(builder)
  ```

- If you want to submit to the daemon (what is done in the example above), you can use instead:

```
from aiida.engine import submit
calcjobnode = submit(builder)
```

  In this case, you just get immediately a reference to the `CalcJobNode` (not yet completed) and you can continue with the execution of the python script. This is useful e.g. if you want to submit many independent calculations at once in a `for` loop, and then let them run in parallel.

From calculations to workflows

AiiDA can help you run individual calculations but it is really designed to help you run workflows that involve several calculations, while automatically keeping track of the provenance for full reproducibility.

As the final step, we are going to launch the `MinimalW90WorkChain` workflow, a demo workflow shipped with the `aiida-wannier90` plugin, that also takes care of running the preliminary DFT steps using Quantum ESPRESSO.

Here is a submission script that we are going to use:

```
from aiida.engine import run
from aiida.orm import Str, Dict, KpointsData, StructureData, load_code
from aiida.plugins import WorkflowFactory

from aiida_wannier90.orbitals import generate_projections


pw_code=load_code("<CODE LABEL>")  # Replace with the QE pw.x code label
wannier_code=load_code("<CODE LABEL>")  # Replace with the Wannier90 wannier.x code␣
↪label
pw2wannier90_code=load_code("<CODE LABEL>")  # Replace with the QE pw2wannier90.x␣
↪code label
pseudo_family_name="<UPF FAMILY NAME>" # Replace with the name of the pseudopotential␣
↪family for SSSP efficiency


# GaAs structure
a = 5.68018817933178  # angstrom
structure = StructureData(
    cell=[[-a / 2., 0, a / 2.], [0, a / 2., a / 2.], [-a / 2., a / 2., 0]]
)
structure.append_atom(symbols=['Ga'], position=(0., 0., 0.))
structure.append_atom(symbols=['As'], position=(-a / 4., a / 4., a / 4.))

# 4x4x4 k-points mesh for the SCF
kpoints_scf = KpointsData()
kpoints_scf.set_kpoints_mesh([4, 4, 4])

# 10x10x10 k-points mesh for the NSCF/Wannier90 calculations
kpoints_nscf = KpointsData()
kpoints_nscf.set_kpoints_mesh([10, 10, 10])

# k-points path for the band structure
kpoint_path = Dict(dict={
    'point_coords': {
        'GAMMA': [0.0, 0.0, 0.0],
        'K': [0.375, 0.375, 0.75],
        'L': [0.5, 0.5, 0.5],
```

(continues on next page)

(continued from previous page)

```
        'U': [0.625, 0.25, 0.625],
        'W': [0.5, 0.25, 0.75],
        'X': [0.5, 0.0, 0.5]
    },
    'path': [('GAMMA', 'X'), ('X', 'U'), ('K', 'GAMMA'),
             ('GAMMA', 'L'), ('L', 'W'), ('W', 'X')]
})

# sp^3 projections, centered on As
projections = generate_projections(
    [
        {
            'position_cart' :(-a / 4., a / 4., a / 4.),
            'ang_mtm_l_list': -3,
            'spin': None,
        },
    ],
    structure=structure
)

# Load the workflow
MinimalW90WorkChain = WorkflowFactory('wannier90.minimal')

# Run the workflow
run(
    MinimalW90WorkChain,
    pw_code=pw_code,
    wannier_code=wannier_code,
    pw2wannier90_code=pw2wannier90_code,
    pseudo_family=Str(pseudo_family_name),
    structure=structure,
    kpoints_scf=kpoints_scf,
    kpoints_nscf=kpoints_nscf,
    kpoint_path=kpoint_path,
    projections=projections
)
```

Download the `demo_bands.py` snippet. You will need to edit the first lines, specifying the name of the AiiDA codes for Quantum ESPRESSO executables `pw.x` and `pw2wannier90.x`, and for the Wannier90 code (that you can discover as usual with `verdi code list`).

Moreover, you need to specify which pseudopotentials you want to use. AiiDA comes with tools to manage pseudopotentials in UPF format (the format used by Quantum ESPRESSO and a few more codes), and to group them in "pseudopotential families". You can list all existing ones with `verdi data upf listfamilies`. We want to use the SSSP library version 1.1. Find the family you want to use and specify its name in the appropriate variable.

Exercise: Inspect the rest of the script to see how we are specifying inputs. In particular:

- Note how you can define a crystal structure directly in AiiDA, without using any external library, if you know the cell vectors and the atoms coordinates.
- Note how you can define a `KpointsData` node (`kpoints_scf`) that does not include an explicit list of k-points, but rather represents a regular grid ($4 \times 4 \times 4$ in this example, and similarly $10 \times 10 \times 10$ for `kpoints_nscf`). The workflow, internally, knows that for the NSCF step one needs to unfold this k-points grid in an explicit grid of all k-points.
- Note a different way to specify the projections, using a more declarative format as a list of projection declarations (that gets internally converted in a queryable `OrbitalData`, rather than just a list of strings). The

documentation of the `get_projections` function and its parameters can be found here.

Once you have saved your changes, you can run the workflow with:

```
verdi run demo_minimal_w90_workchain.py
```

This workflow will:

1. Run a SCF simulation on GaAs
2. Run an NSCF calculation on a denser grid (with an explicit list of k-points)
3. Run a post-process Wannier90 `-pp` calculation to get the `.nnkp` file
4. Run the interface code `pw2wannier90.x`
5. Run the Wannierisation step, returning the interpolated bands.

The workflow should take ~5 minutes on a typical laptop. You may notice that `verdi process list` now shows more than one entry. While you wait for the workflow to complete, let's start exploring its provenance.

The full provenance graph obtained from `verdi node graph generate` will already be rather complex (you can try!), so let's try browsing the provenance interactively instead.

In a new verdi shell, start the AiiDA REST API:

```
verdi restapi
```

and open the (from the browser inside the virtual machine).

Note: The provenance browser is a Javascript application provided by Materials Cloud that connects to your AiiDA REST API. *Your data never leave your computer*.

Browse your AiiDA database.

- Start by finding your workflow (in the left menu, filter the nodes by selecting `Process -> Workflow -> WorkChain`. WorkChains are a specific type of workflows in AiiDA that allow to define multiple steps and can be paused and restarted between steps).
- Inspect the inputs and returned outputs of the workflow in the provenance browser (outputs will appear once the calculations are done). Moreover, you can inspect the calculations that the workflow launched, and check both their input and output nodes (via the provenance browser), as well as the raw inputs and outputs of the calculation (in the page of a specific `CalcJobNode`).

Note: When perfoming calculations for a research project, at the end upon publication you can export your provenance graph using `verdi export create` and upload it to the Materials Cloud Archive, enabling your peers to explore the provenance of your calculations.

Once the workchain is finished, use `verdi process show <PK>` to inspect the `MinimalW90WorkChain` and find the PK of its `interpolated_bands` output. Use this to produce an `xmgrace` output of the interpolated band structure:

```
verdi data bands export --format agr --output wannier_bands.agr <PK>
```

that you can then visualise using `xmgrace wannier_bands.agr`.

How to continue from here

The following appendix to this chapter discusses a bit more in detail how to import and export crystal structures from an AiiDA database, and which tools exist to obtain the primitive structure from a conventional structure, or more generally from a supercell.

Importantly, the conversion of a crystal structure (conventional cell) to a primitive cell means the creation of a Data node from another Data node using a python function. The appendix shows how easy it is with AiiDA to wrap python functions into AiiDA's `calcfunctions` that automatically preserve the provenance of the data transformation in the AiiDA graph.

If you do not have time right now, you can *jump directly to the next tutorial section*, where we will see how to use fully automated AiiDA workflows that can obtain Wannier functions of a material with minimal input, without the need to specifying input parameters apart from the crystal structure and the code to run (and where even the choice of initial projections is automated).

Appendix: Get the primitive structure while preserving the provenance

Let us begin by downloading a gallium arsenide (GaAs) structure from the Crystallography Open Database and importing it into AiiDA.

Note: You can view the structure online.

```
wget http://crystallography.net/cod/9008845.cif
verdi data structure import ase 9008845.cif
```

As before, you will get the `PK` of the imported structure. We note that a `StructureData` can also be exported to file to various formats. As an example, let's export the structure in XSF format and visualize it with XCrySDen:

```
verdi data structure export --format=xsf <PK> > exported.xsf
xcrysden --xsf exported.xsf
```

You should see the GaAs supercell (8 atoms) that we downloaded from the COD database (in CIF format), imported into AiiDA and exported back into a different format (XSF).

In particular, the structure that we imported (and converted from CIF to an explicit list of atoms) used the conventional cell with 8 atoms, rather than the primitive one (with 2 atoms only).

Getting the primitive structure

We will now use seekpath (that internally uses spglib), to obtain the primitive cell from the conventional one. Actually, seekpath does more: it will also standardise the orientation of the structure, and suggest the path for a band-structure calculation.

We will not use seekpath directly, but a wrapper for AiiDA, that converts to and from AiiDA data types automatically (in particular, it reads directly AiiDA data nodes, and returns AiiDA nodes). The full documentation of these wrapper methods can be found on this page.

As a first thing, in the terminal, open an ipython shell with the AiiDA environment pre-loaded. This can be achieved by running:

```
verdi shell
```

Note: If you prefer working in jupyter, you can do so.

Open your browser (the one with jupyter that you opened in the *setup* chapter of this tutorial), then create a new Python 3 notebook (with the button "New" in the top right of the jupyter page, and then select "Python 3").

Give the notebook a name: click on the word "Untitled" at the top, then type a file name (e.g. "create_supercell") and confirm.

The only additional thing you need to do is to load the AiiDA environment. In the first code cell, type `%aiida` and confirm with Shift+ENTER.

You should get a confirmation message "Loaded AiiDA DB environment". You can then work as usual in jupyter, adding the code in the following cells.

Now, in this ipython shell (or in jupyter), you can import the wrapper function (that internally uses seekpath) as:

```
from aiida.tools import get_kpoints_path
```

We now want to use it. As you can see in the documentation, you need to pass as a parameter a `StructureData` node (in our case, the node that you imported earlier from COD). To load a node in the ipython shell, use the following command:

```
structure = load_node(<PK>)
```

where `<PK>` is the of the `StructureData` node you imported earlier. At this point you are ready to get the primitive structure:

```
seekpath_data = get_kpoints_path(structure)
primitive = seekpath_data['primitive_structure']
```

Exercise: Check that the `primitive` object is an AiiDA `StructureData` node, and that it is still not stored in the database (you can just print it). Moreover, check that you indeed obtained a primitive structure by inspecting the unit cell using `print(primitive.cell)`, and the list and position of the atoms with `print(primitive.sites)`.

Preserving the provenance

AiiDA is focused on making it easy to track the provenance of your calculation, i.e., the history of how it has been generated, by which calculation, and with which inputs. If you were just to store the `primitive StructureData` node, you would lose its provenance. Instead, AiiDA provides simple tools to store it automatically in the form of a graph, where nodes are either data nodes (as the one we have just seen), or calculations (i.e., "black boxes" that get data as input, and create new data as output). Links between nodes represent the logical relationship between calculations and their inputs and outputs.

While we refer to the full AiiDA documentation for more in-depth explanations, here we show the simplest way to run a python function while keeping the provenance at the same time.

This can be achieved by using a `calcfunction`: this is a wrapper around python functions (technically, a python function decorator) that takes care of storing the execution of that function in the graph. To use it, you need first to create a simple function that gets one or more AiiDA nodes, and returns one AiiDA node (or a dictionary of AiiDA nodes). Moreover, you need to decorate it as a `calcfunction`, so that when it will be run, it will be stored in the database.

Here is the complete code:

```
from aiida.engine import calcfunction

@calcfunction
def get_primitive(input_structure):
    from aiida.tools import get_kpoints_path
    seekpath_data = get_kpoints_path(input_structure)
    return {
        'primitive_structure': seekpath_data['primitive_structure'],
        'seekpath_parameters': seekpath_data['parameters']
    }
```

Once you have defined the function, run it on the `structure` node you loaded earlier:

```
results = get_primitive(structure)
primitive = results['primitive_structure']
```

The first thing you can notice (by printing `primitive`) is that now this node has been automatically stored. Additionally, you can check who created it simply as `creator_function = primitive.creator`. You can for instance check the name of the function that was run, using `creator_function.attributes['function_name']` (this returns `get_primitive`, the name of the function that we decorated as a `calc_function`). Moreover, you can check the inputs of this function.

Exercise: Use `creator_function.inputs.input_structure` to get the input of the function called `input_structure` (the name is taken from the parameter name in the definition of the `get_primitive` function) and check that it is the exact same node that you started from.

Exercise: Go in a bash shell (e.g., open a new terminal – remember to also enter the virtual environment using `workon aiida`!). You can inspect graphically the full provenance of a given node using

```
verdi node graph generate <PK>
```

where we suggest here to use the PK of the `creator_function`. The code will generate a PDF that you can open, and that should look like the following image:

Note: TAKE HOME MESSAGE

AiiDA makes it very easy to convert python functions into `calcfunctions` that, every time they are executed, represent their execution in the AiiDA graph.

The calc function itself is represented as a Calculation Node; all its function inputs and its (labeled) function outputs are also stored as AiiDA data nodes and are linked via INPUT and CREATE links, respectively.

It is possible to browse such graph, that tracks the *provenance* of the output data. Moreover, having the exact inputs tracked makes each calculation *reproducible*.

Finally (as we have already seen as an example in the previous sections for calculation jobs), outputs of a calculation can become inputs to a new calculation. Therefore, the AiiDA data provenance graph is a **directed acyclic graph**.

Note: While you can run the `get_kpoints_path` function as many times you want (and it will return unstored nodes, so it will not clutter your AiiDA database), remember that *every time* you run the `get_primitive()` calc function, you will get a bunch of new nodes in the database automatically stored for you.

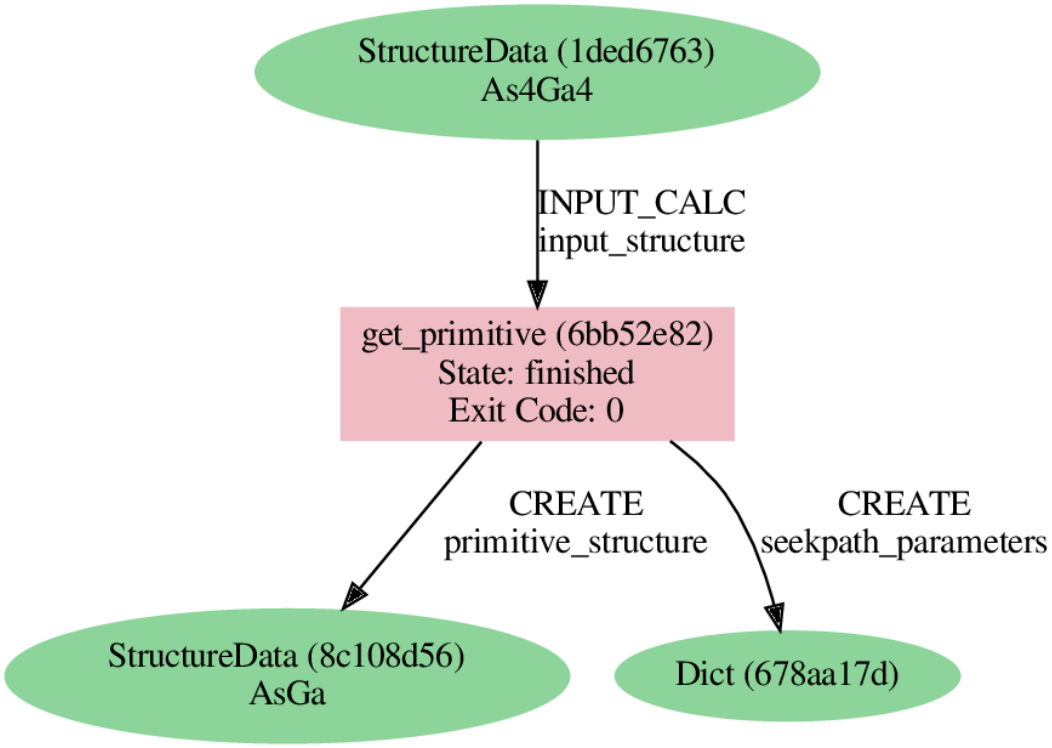


Fig. 1.3: Provenance graph for the calcfunction used to obtain the primitive structure of GaAs.

Automated high-throughput wannierisation

In the following tutorial you will learn how to perform automated high-throughput Wannierisation using a dedicated AiiDA workchain.

The protocol for automating the construction of Wannier functions is discussed in the following article:

- Valerio Vitale, Giovanni Pizzi, Antimo Marrazzo, Jonathan Yates, Nicola Marzari, Arash Mostofi, *Automated high-throughput wannierisation*, accepted in npj Computational Materials (2020); https://arxiv.org/abs/1909.00433

whose data is available on the Materials Cloud Archive as:

- Valerio Vitale, Giovanni Pizzi, Antimo Marrazzo, Jonathan R. Yates, Nicola Marzari, Arash A. Mostofi, Automated high-throughput Wannierisation, Materials Cloud Archive (2019), doi: 10.24435/materialscloud:2019.0044/v2.

The protocol leverages the SCDM method introduced in:

- Anil Damle, Lin Lin, and Lexing Ying, Compressed representation of kohn–sham orbitals via selected columns of the density matrix Journal of Chemical Theory and Computation 11, 1463–1469 (2015).
- Anil Damle and Lin Lin, Disentanglement via entanglement: A unified method for wannier localization, Multiscale Modeling & Simulation 16, 1392–1410 (2018).

The initial workflow was written by Antimo Marrazzo (EPFL) and Giovanni Pizzi (EPFL) (and is available, in a Virtual Machine, on the Materials Cloud entry mentioned above). It was later substantially improved and upgraded to AiiDA v1.x by Junfeng Qiao (EFPL). The SCDM implementation in Quantum ESPRESSO was done by Valerio Vitale (Imperial College London and University of Cambridge).

Note: Launch while you read! The workflow should take a few minutes to run on the virtual machine (5-10 minutes). So, we suggest that you launch it now (following the suggestions below) in the background, while you read through the tutorial.

Preparation

You can download the launch script for the workflow here: `launch_auto-wannier_workflow.py`. The script accepts one argument, which is the location of the crystal structure file (in XSF format) for which you want to run the Wannierisation. As an example, you can pick any one of the simple crystal structures from this list below:

- `Ar2.xsf`
- `BrNa.xsf`
- `GaAs.xsf`
- `F4Ni2.xsf`
- `O2Rb2.xsf`
- `BaS.xsf`
- `C6Mg4.xsf`
- `FNa.xsf`
- `O2Sr.xsf`
- `BeO4S.xsf`
- `CaO.xsf`

- `Cr2F4.xsf`
- `O2Pb2.xsf`
- `PtS2.xsf`
- `Br2Ti.xsf`
- `Cl2O2Ti2.xsf`
- `CsH.xsf`
- `O2Pd2.xsf`

Once you downloaded both the launcher script and one of the XSF files, first adapt the launcher script content, as usual, filling in the names of the codes that you want to use.

Then, launch the script with the following command:

```
verdi run launch_auto-wannier_workflow.py CsH.xsf
```

(in the following, we will use CsH as an example); you can replace `CsH.xsf` with any other structure of the list above, e.g. `PtS2.xsf`, or `Br2Ti.xsf`, ...

Introduction

The use of maximally-localised Wannier functions (MLWFs) in high-throughput (HT) workflows has been hindered by the fact that generating MLWFs automatically and robustly without any user intervention and for arbitrary materials is, in general, very challenging.

The procedure to obtain MLWFs requires to specify a number of parameters that depends on the specific system under study, such as

- the type of projections to build the $A_{mn}(\mathbf{k})$ matrix

and, in the case of entangled bands, many other parameters like

- the number of Wannier functions,
- the frozen energy window,
- the outer energy window, ...

The SCDM method allows to circumvent the need to specify initial projections, by introducting an algorithm that builds the $A_{mn}(\mathbf{k})$ matrix by optimally selecting columns of the density matrix $P_{\mathbf{k}}$

$$P_{\mathbf{k}}(\mathbf{r},\mathbf{r}') = \sum_{n=1}^{J} f(\epsilon_{n\mathbf{k}})\psi_{n\mathbf{k}}(\mathbf{r})\psi^*_{n\mathbf{k}}(\mathbf{r}')$$

where $f(\epsilon_{n\mathbf{k}})$ is an occupation function. The SCDM algorithm is based on a QR factorization with column pivoting (QRCP) and it is currently implemented in the `pw2wannier90.x` code of Quantum Espresso.

It is worth to stress that the occupation function does not necessarily correspond to a physical smearing, but it is used as a "window function" that restricts the manifold to the energy region of interest. For example, the isolated-band case can be recovered by setting $f(\epsilon_{n\mathbf{k}}) = 1$ for energy values $\epsilon_{n\mathbf{k}}$ within the energy range of the isolated bands, and zero elsewhere.

Another typical choice for the occupation function is the so-called *complementary error function* (`erfc`)

$$f(\epsilon) = \frac{1}{2}\mathrm{erfc}\left(\frac{\epsilon-\mu}{\sigma}\right)$$

and it is used to to deal with a manifold of bands that are entangled only in the upper energy region (e.g. for metals, or for the conduction band of insulators). Another possible choice for the occupation function could be a Gaussian, for instance to extract bands in a specific energy range from a fully entangled manifold.

While the SCDM method allows to avoid the disentanglement procedure, and so the need to specify the frozen and outer window, it does not provide a recipe to set the smearing function parameters μ and σ. In addition, the number of Wannier functions remains to be set and a sensible value typically requires some chemical consideration, such as counting the number of atomic orbitals of a given orbital character (e.g. s, p, d, sp^3, ...).

The `Wannier90BandsWorkChain` (distributed in the aiida-wannier90-workflows package) is an AiiDA workchain that implements a protocol that deals with the choice of the number of Wannier functions and sets the parameters μ and σ defining the smearing function. For a full explanation of the protocol we refer to the article Automated high-throughput wannierisation, while here we just outline the main features.

The workflow starts by running a DFT calculation (with Quantum ESPRESSO) using automated workchains distributed within the aiida-quantumespresso plugin, that take care of automation for the calculations performed with Quantum ESPRESSO (QE). This is followed by a calculation of the *projectability*, that is the projection of the Kohn-Sham states onto localised atomic orbitals:

$$p_{n\mathbf{k}} = \sum_{I,l,m} |\langle \psi_{n\mathbf{k}} | \phi^{\mathbf{k}}_{Ilm} \rangle|^2,$$

where the $\phi_{Ilm}(\mathbf{k})$ are the pseudo-atomic orbitals (PAO) employed in the generation of the pseudopotentials, I is an index running over the atoms in the cell and lm define the usual angular momentum quantum numbers.

The workflow is designed for the specific use case where we are interested in wannierising the occupied bands (plus, optionally, some unoccupied or partially occupied bands) in insulators and in metals.

The **number of Wannier functions** is automatically set equal to the number of PAOs defined in the pseudopotentials (and the pseudopotentials are automatically taken from the SSSP efficiency library). The projectabilities on these PAO orbitals are then computed and used to set the **optimal smearing function (erfc) parameters** μ and σ, as explained in the paper Automated high-throughput wannierisation. After the calculation of the projectabilities, the workflow proceeds with the usual Wannierisation step: first it computes the overlap and projection matrices using `pw2wannier90`, and then it runs the Wannier90 code.

Here we summarise the main steps of the `Wannier90BandsWorkChain`:

- SCF (QuantumESPRESSO `pw.x`)
- NSCF (QuantumESPRESSO `pw.x`)
- Projectability (QuantumESPRESSO `projwfc.x`)
- Wannier90 pre-processing (`wannier90.x -pp`)
- Overlap matrices M_{mn}, initial projections with SCDM A_{mn} (QuantumESPRESSO `pw2wannier90.x`)
- Wannierisation (`wannier90.x`)

The output of the workflow includes several nodes, among which the projectabilities and interpolated band structure, that we are going to inspect after the run.

Running the workflow

If you have not launch the script yet, please do it now!

Here we focus on how to run the `Wannier90BandsWorkChain`, the AiiDA workchain that implements the automation workflow to obtain MLWFs.

Important note: For this tutorial, in order to keep the total simulation time down to ~5-10 minutes on a typical laptop, we run the workflow using the `testing` peotocol, *where all the wavefunction cutoffs are halved to speed up the*

calculations (with respect to the converged and tested cutoffs suggested by the SSSP pseudopotential library). For production please use the `theos-ht-1.0` protocol (or any other protocol with converged cutoffs).

We remind you that, to get a list of all the AiiDA processes (including calculations and workchains) that are running and their status you can use:

```
verdi process list
```

or alternatively

```
verdi process list -p1 -a
```

to see *all* workflows and calculations, even completed, that have been launched in the past 1 day.

Since you have already run the workflow by now, run the commands above to retrieve the corresponding PKs.

Here is the script that you have run:

```
#!/usr/bin/env runaiida
import sys
from aiida.orm import StructureData, Bool, Code, Dict
from aiida.engine import submit
from ase.io import read as aseread
from aiida_wannier90_workflows.workflows import Wannier90BandsWorkChain

# Codenames for pw.x, pw2wannier90.x, projwfc.x and wannier90.x
# Please modify these according to your machine
pw_code = Code.get_from_string("<CODE LABEL>")  # e.g. 'qe-6.5-pw@localhost'
pw2wan_code = Code.get_from_string("<CODE LABEL>")  # e.g. 'qe-6.5-
↪pw2wannier90@localhost'
projwfc_code = Code.get_from_string("<CODE LABEL>")  # e.g. 'qe-6.5-projwfc@localhost'
wan_code = Code.get_from_string("<CODE LABEL>")  # e.g. 'wannier90-3.1.0-
↪wannier@localhost'

# The 1st commandline argument specifies the structure to be calculated
xsf_file = sys.argv[1]  # e.g. 'CsH.xsf'

# Read xsf file and convert into a stored StructureData
structure = StructureData(ase=aseread(xsf_file))

# Prepare the builder to launch the workchain
builder = Wannier90BandsWorkChain.get_builder()
builder.structure = structure
builder.code = {
    'pw': pw_code,
    'pw2wannier90': pw2wan_code,
    'projwfc': projwfc_code,
    'wannier90': wan_code
}
# For this tutorial, we are using the 'testing' protocol,
# with all cutoffs halved, to speed up the simulations
builder.protocol = Dict(dict={'name': 'testing'})
# Flags to control the workchain behaviour
builder.controls = {
    # If True, compute only valence bands (NB: use for insulators only!)
    'only_valence': Bool(False),
    # If True, perform maximal-localisation (MLWF) procedure, i.e., minimise the␣
↪spread Omega
    'do_mlwf': Bool(True),
```

(continues on next page)

(continued from previous page)

```
}

# Submits the workchain
workchain = submit(builder)

print('launched WorkChain<{}> for structure {}'.format(workchain.pk, structure.get_
↪formula()))
print("Use `verdi process list` or `verdi process show {}` to check the progress".
↪format(workchain.pk))
```

Inspect the script in detail now, and make sure that you understand all its parts. The comments should guide you through it. As you will notice, the amount of information to provide to the workflow is really minimal: just the codes to use, the crystal structure, and some flags to control the behavior (which protocol to use, if you want to compute only valence bands, and if you want to run the MLWF step).

Analyzing the outputs of the workflow

Now we analyse the reports and outputs of the workflow using the command line.

Both while the `Wannier90BandsWorkChain` is running, and after its completion, you can monitor its progress by looking at its "report", using the command

```
verdi process report <PK>
```

where `<PK>` corresponds to the workchain PK. You will see a list of log with messages produced by the workchain, including the PKs of all its sub-workchains and of the calculations launched by the `Wannier90BandsWorkChain`. The report will be similar to the following:

```
2020-03-19 07:58:27 [88  | REPORT]: [495|Wannier90BandsWorkChain|setup_protocol]:␣
↪running the workchain with the "testing" protocol
2020-03-19 07:58:27 [89  | REPORT]: [495|Wannier90BandsWorkChain|setup]: workchain␣
↪controls found in inputs: valence + conduction bands
2020-03-19 07:58:27 [90  | REPORT]: [495|Wannier90BandsWorkChain|run_seekpath]:␣
↪running seekpath to get primitive structure for: CsH
2020-03-19 07:58:28 [91  | REPORT]: [495|Wannier90BandsWorkChain|setup_parameters]:␣
↪number of machines 1 auto-set according to number of atoms
2020-03-19 07:58:30 [92  | REPORT]: [495|Wannier90BandsWorkChain|run_wannier_
↪workchain]: launching Wannier90WorkChain<514>
2020-03-19 07:58:31 [93  | REPORT]:   [514|Wannier90WorkChain|run_scf]: scf step -␣
↪launching PwBaseWorkChain<517> in scf mode
2020-03-19 07:58:33 [94  | REPORT]:     [517|PwBaseWorkChain|run_calculation]:␣
↪launching PwCalculation<523> iteration #1
2020-03-19 07:58:46 [95  | REPORT]:     [517|PwBaseWorkChain|inspect_calculation]:␣
↪PwCalculation<523> completed successfully
2020-03-19 07:58:46 [96  | REPORT]:     [517|PwBaseWorkChain|results]: work chain␣
↪completed after 1 iterations
2020-03-19 07:58:46 [97  | REPORT]:     [517|PwBaseWorkChain|on_terminated]: remote␣
↪folders will not be cleaned
2020-03-19 07:58:47 [98  | REPORT]:   [514|Wannier90WorkChain|inspect_scf]: scf␣
↪PwBaseWorkChain successfully finished
2020-03-19 07:58:47 [99  | REPORT]:   [514|Wannier90WorkChain|run_nscf]: nscf number␣
↪of bands set as 24
2020-03-19 07:58:48 [100 | REPORT]:   [514|Wannier90WorkChain|run_nscf]: nscf step -␣
↪launching PwBaseWorkChain<537> in nscf mode
2020-03-19 07:58:49 [101 | REPORT]:     [537|PwBaseWorkChain|run_calculation]:␣
↪launching PwCalculation<540> iteration #1
```

(continues on next page)

(continued from previous page)

```
2020-03-19 08:01:16 [102 | REPORT]:     [537|PwBaseWorkChain|inspect_calculation]:␣
↪PwCalculation<540> completed successfully
2020-03-19 08:01:16 [103 | REPORT]:     [537|PwBaseWorkChain|results]: work chain␣
↪completed after 1 iterations
2020-03-19 08:01:16 [104 | REPORT]:     [537|PwBaseWorkChain|on_terminated]: remote␣
↪folders will not be cleaned
2020-03-19 08:01:17 [105 | REPORT]:   [514|Wannier90WorkChain|inspect_nscf]: nscf␣
↪PwBaseWorkChain successfully finished
2020-03-19 08:01:17 [106 | REPORT]:   [514|Wannier90WorkChain|should_do_projwfc]:␣
↪SCDM mu & sigma are auto-set using projectability
2020-03-19 08:01:18 [107 | REPORT]:   [514|Wannier90WorkChain|run_projwfc]: projwfc␣
↪step - launching ProjwfcCalculation<546>
2020-03-19 08:01:27 [108 | REPORT]:   [514|Wannier90WorkChain|inspect_projwfc]:␣
↪projwfc ProjwfcCalculation successfully finished
2020-03-19 08:01:28 [109 | REPORT]:   [514|Wannier90WorkChain|run_wannier90_pp]:␣
↪number of Wannier functions extracted from projections: 14
2020-03-19 08:01:29 [110 | REPORT]:   [514|Wannier90WorkChain|run_wannier90_pp]:␣
↪wannier90 postproc step - launching Wannier90Calculation<559> in postproc mode
2020-03-19 08:01:35 [111 | REPORT]:   [514|Wannier90WorkChain|inspect_wannier90_pp]:␣
↪wannier90 postproc Wannier90Calculation successfully finished
2020-03-19 08:01:38 [112 | REPORT]:   [514|Wannier90WorkChain|run_pw2wannier90]:␣
↪pw2wannier90 step - launching Pw2Wannier90Calculation<567>
2020-03-19 08:02:16 [113 | REPORT]:   [514|Wannier90WorkChain|inspect_pw2wannier90]:␣
↪Pw2wannier90Calculation successfully finished
2020-03-19 08:02:16 [114 | REPORT]:   [514|Wannier90WorkChain|run_wannier90]:␣
↪wannier90 step - launching Wannier90Calculation<572>
2020-03-19 08:03:06 [115 | REPORT]:   [514|Wannier90WorkChain|inspect_wannier90]:␣
↪Wannier90Calculation successfully finished
2020-03-19 08:03:06 [116 | REPORT]:   [514|Wannier90WorkChain|results]: final step -␣
↪preparing outputs
2020-03-19 08:03:06 [117 | REPORT]:   [514|Wannier90WorkChain|results]:␣
↪Wannier90WorkChain successfully completed
2020-03-19 08:03:07 [118 | REPORT]: [495|Wannier90BandsWorkChain|results]: wannier90␣
↪interpolated bands pk: 575
2020-03-19 08:03:07 [119 | REPORT]: [495|Wannier90BandsWorkChain|results]:␣
↪Wannier90BandsWorkChain successfully completed
```

Once the workchain has finished to run, you can look at all its inputs and outputs with

```
verdi node show <PK>
```

You should obtain an output similar to what follows:

```
Property     Value
-----------  ------------------------------------
type         Wannier90BandsWorkChain
state        Finished [0]
pk           404
uuid         df46175b-4634-4edf-a303-e90af98a27fc
label
description
ctime        2020-03-18 18:38:36.294955+00:00
mtime        2020-03-18 18:50:59.245496+00:00
computer     [1] localhost

Inputs                          PK    Type
------------------------------  ----  -------------
```

(continues on next page)

(continued from previous page)

```
code
    wannier90                    17    Code
    projwfc                      6     Code
    pw2wannier90                 7     Code
    pw                           1     Code
controls
    kpoints_distance_for_bands   401   Float
    do_disentanglement           400   Bool
    plot_wannier_functions       399   Bool
    retrieve_hamiltonian         398   Bool
    auto_projections             397   Bool
    do_mlwf                      396   Bool
    only_valence                 395   Bool
protocol                         403   Dict
structure                        402   StructureData

Outputs                          PK  Type
-----------------------------  ----  --------------
nscf_parameters                454  Dict
primitive_structure            409  StructureData
projwfc_bands                  460  BandsData
projwfc_projections            459  ProjectionData
pw2wannier90_remote_folder     477  RemoteData
scf_parameters                 437  Dict
seekpath_parameters            407  Dict
wannier90_interpolated_bands   484  BandsData
wannier90_parameters           485  Dict
wannier90_remote_folder        482  RemoteData
wannier90_retrieved            483  FolderData

Called      PK  Type
--------  ----  ----------------
CALL       423  WorkChainNode
CALL       406  CalcFunctionNode

Log messages
--------------------------------------------
There are 7 log messages for this calculation
Run 'verdi process report 404' to see them
```

Analyzing and comparing the band structure

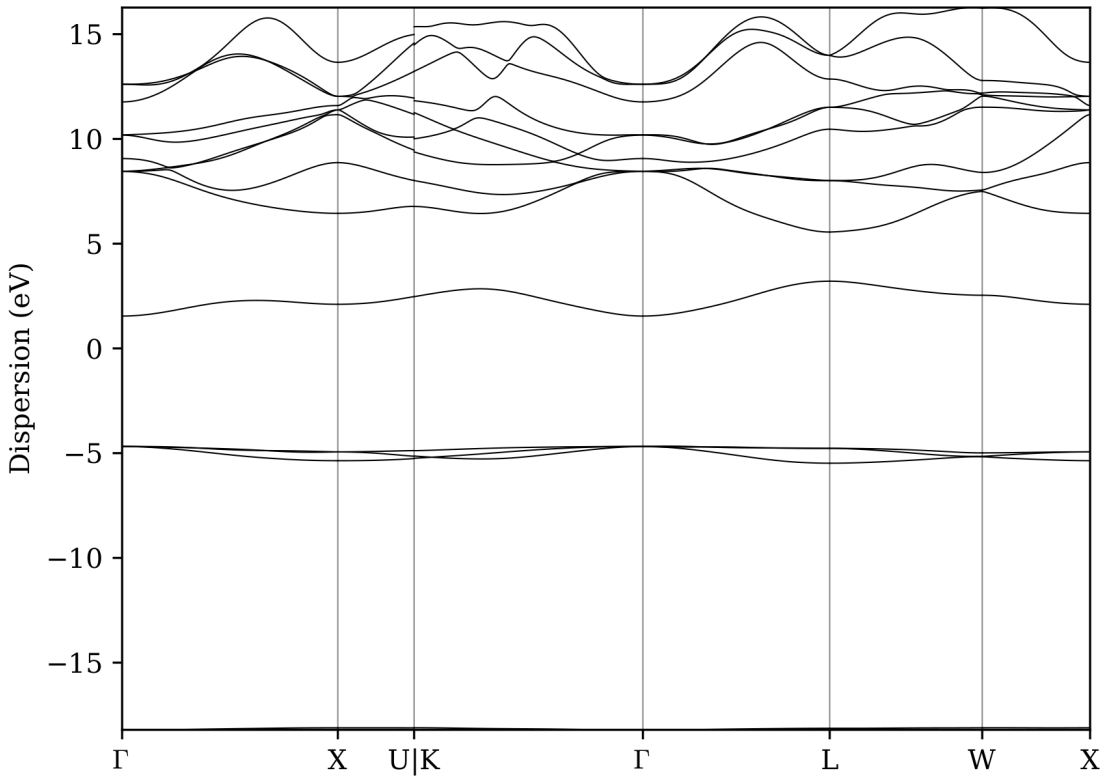
First let's give a look at the interpolated band structure by exporting it to a PDF file with

```
verdi data bands export --format mpl_pdf --output band_structure.pdf <PK_bands>
```

where `<PK_bands>` stands for the `BandsData` PK produced by the workflow. You can find it from the output of the `verdi node show <PK>` command that you run before. You should obtain a PDF like the following:

Now we compare the Wannier-interpolated bands with the full DFT bands calculation. For convenience, we have already computed for you all the full DFT band structures for all the compounds for which we provided the XSF above (even if you could easily recompute these band structures using the plugins and workflows included in the `aiida-quantumespresso` package). You can download the full DFT bands in xmgrace format (`.agr`) form this list:

- `Ar2_dft_bands.agr`


15
10
5
0
−5
−10
−15
Dispersion (eV)
Γ
X
U|K
Γ
L
W
X

- `BrNa_dft_bands.agr`
- `AsGa_dft_bands.agr`
- `F4Ni2_dft_bands.agr`
- `O2Rb2_dft_bands.agr`
- `BaS_dft_bands.agr`
- `C6Mg4_dft_bands.agr`
- `FNa_dft_bands.agr`
- `O2Sr_dft_bands.agr`
- `BeO4S_dft_bands.agr`
- `CaO_dft_bands.agr`
- `Cr2F4_dft_bands.agr`
- `O2Pb2_dft_bands.agr`
- `PtS2_dft_bands.agr`
- `Br2Ti_dft_bands.agr`
- `Cl2O2Ti2_dft_bands.agr`
- `CsH_dft_bands.agr`
- `O2Pd2_dft_bands.agr`

In particular, taking CsH as an example, you can first export the Wannier-interpolated bands that you just computed earlier in xmgrace format with

```
verdi data bands export --format agr --output CsH_wan_bands.agr <PK_bands>
```

and then you can compare them with the full DFT band structure using xmgrace typing:

```
xmgrace CsH_dft_bands.agr CsH_wan_bands.agr
```

where you can replace CsH with the correct chemical formula of the crystal structure that you used.

For reference, here are the two xmgrace files for CsH: `CsH_dft_bands.agr` `CsH_wan_bands.agr`.

In the case of CsH, you should obtain something like the following plot (note that we slightly updated the colors and graphical appearance of the plot with respect to the default one you would get by running the command above):

Analyzing the projectabilities

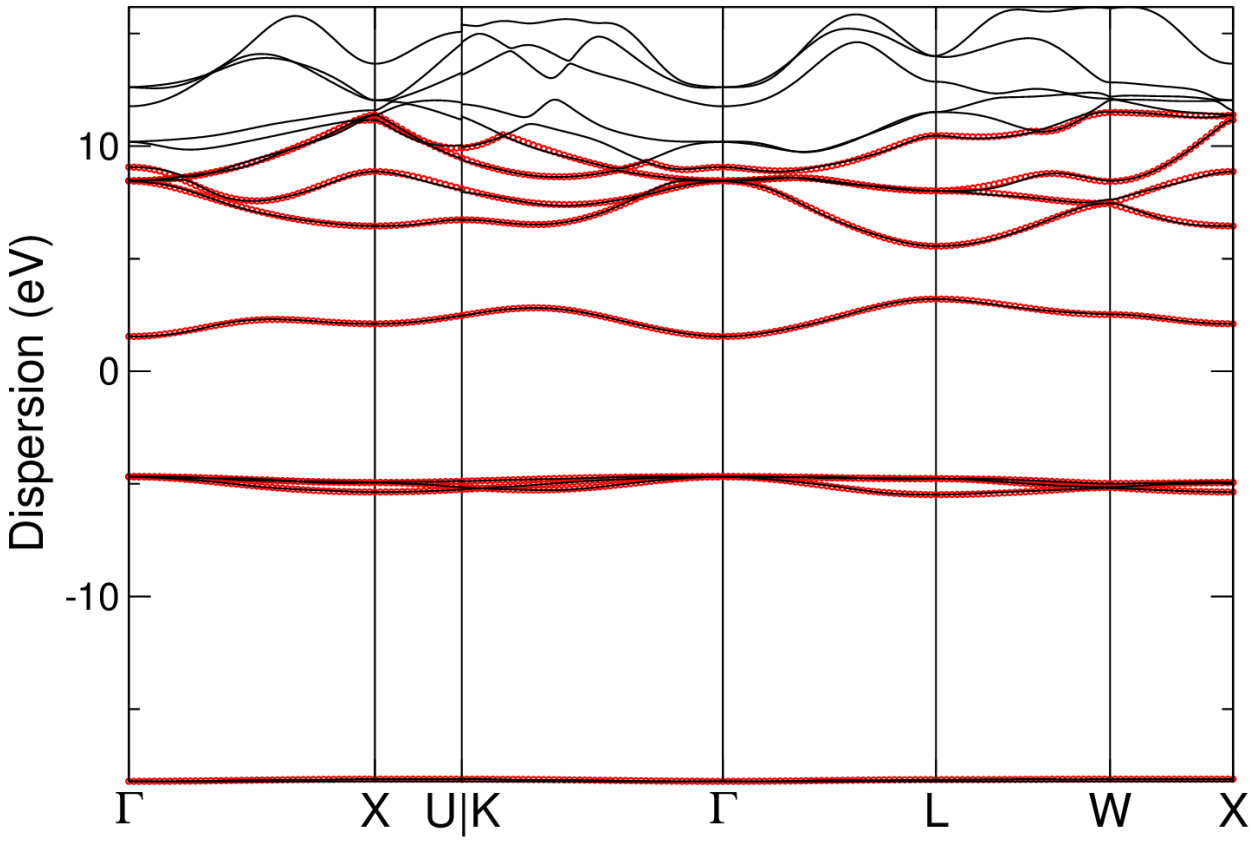
Now you will see how to look at the projectabilities that have been computed by the workchain and then used to obtain μ and σ in the automation protocol. You can download the following script `plot_projectabilities.py` and run it

```
verdi run plot_projectabilities.py <PK>
```

where PK stands for the `Wannier90BandsWorkChain` pk.

You should obtain a plot similar to the following:

Exercise: Open the script and try to understand what it is doing.



As you can see, the protocol to choose μ and σ ensures that the SCDM algorithm is applied to a density-matrix that includes (with a large weight) only those Kohn-Sham states that have a large projection on the manifold spanned by the PAOs.

Analyzing the provenance graph

We begin by generating the provenance graph with

```
verdi node graph generate <PK>
```

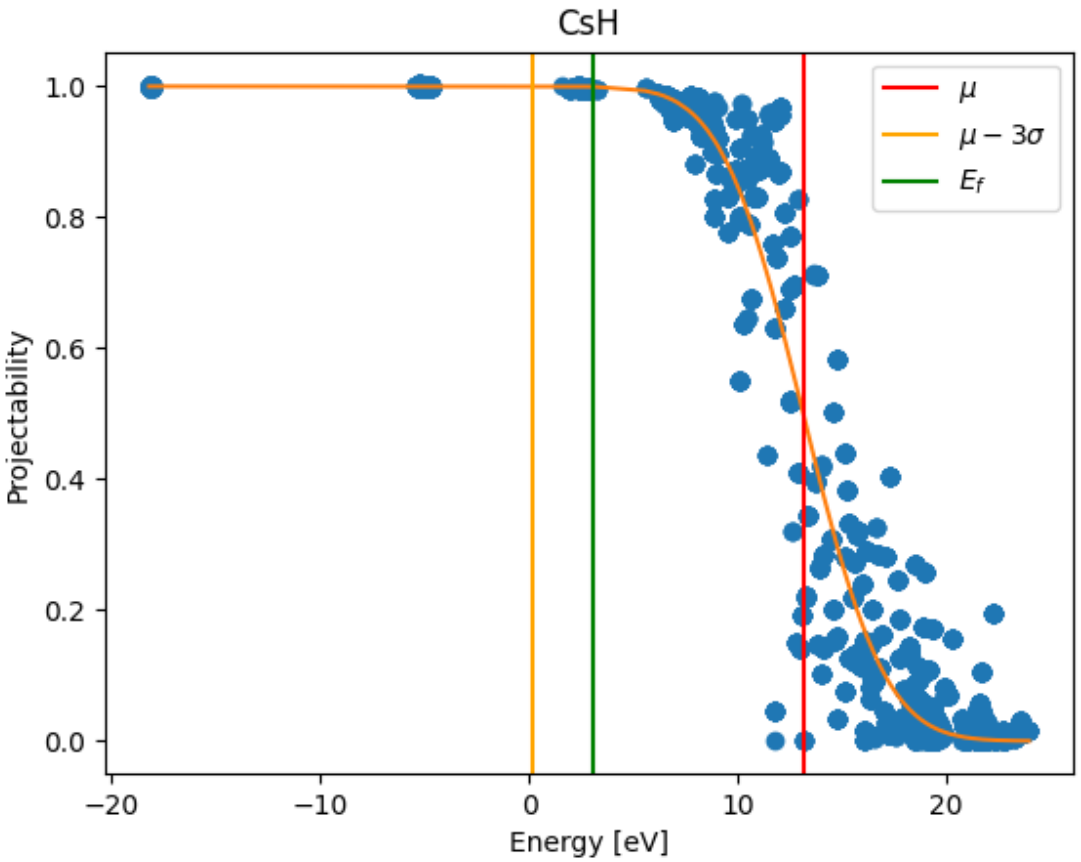
where the PK corresponds to the workflow you have just run. You should obtain something like the following:

As you can see, AiiDA has tracked all the inputs provided to all calculations (and their outputs), allowing you (or anyone else) to reproduce it later on.

(Optional) Maximal localisation and SCDM

Try to modify the `launch_auto-wannier_workflow.py` script to disable the MLWF procedure in order to obtain Wannier functions with SCDM projections that are not maximally localised.

Exercise: Run the workflow for 1 or 2 materials of the dataset. Do you notice any difference when using or not the MLWF procedure? Which one gives better results? Do your results agree with the findings of the Automated high-throughput wannierisation paper?


CsH
Projectability
Energy [eV]
1.0
0.8
0.6
0.4
0.2
0.0
−20
−10
0
10
20
μ
$\mu - 3\sigma$
E_f

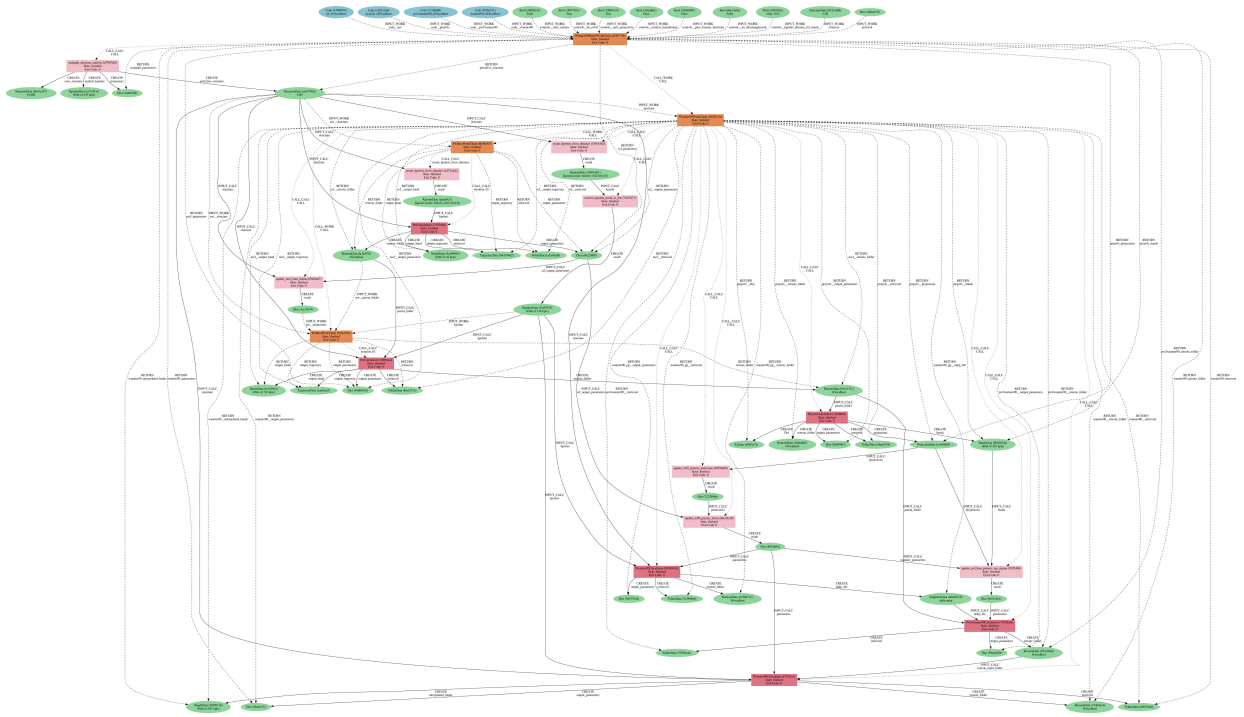


Fig. 1.4: Provenance graph for a single `Wannier90BandsWorkChain` run. (PDF version `CsH.dot.pdf`)

(Optional) Browse your database with the REST API

Connect to the AiiDA REST API and browse your database! Follow the instructions that you find on the Materials Cloud website.

(Optional) More on AiiDA

You now have a first taste of the type of problems AiiDA tries to solve, and you have seen how it is possible, thanks to the tools provided by AiiDA, to develop plugins and workflows to automate your research tasks, while preserving the full provenance and guaranteeing reproducibility of your results.

Here are some options for how to continue:

- Continue with the in-depth tutorial and learn more about the `verdi`, `verdi shell` and `python` interfaces to AiiDA. There is more than enough material to keep you busy for a day. You can check for instance the tutorial held in EPFL in May 2019 (note that this has been tested on a beta version of AiiDA 1.0 and we didn't check yet if anything needs to be adapted for AiiDA 1.1, that you have in your VM).
- **For advanced Linux & python users**: try setting up AiiDA directly on your laptop. Note that AiiDA depends on a number of services and software that require some time to set up. Unfortunately, we will not have the time to help you solve issues related to your setup in a tutorial context, but you can refer to the AiiDA documentation and to the AiiDA mailing list.

1.1.2 In-depth tutorial

In this tutorial, no in-depth tutorial on AiiDA has been presented.

If you are interested, you can check the in-depth tutorial of the Xiamen 2019 tutorial or of the EPFL 2019 tutorial.

CHAPTER

TWO

ADDITIONAL MATERIAL

All the material for all past AiiDA tutorials can be found online on the official AiiDA tutorials page.

The AiiDA cheatsheet, with class hierarchy and common commands, can be probably useful as a quick reference.