AiiDA tutorial

The AiiDA Team

January 2017

Contents

1 Using the verdi command line

This part of the tutorial will help you familiarize with the command line utility `verdi`, one of the most common ways to interact with AiiDA. `verdi` with its subcommands enables a variety of operations such as inspecting the status of ongoing or terminated calculations, showing the details of calculations, computers, codes, or data structures, access the input and the output of a calculation, etc. Similar to the `bash` shell, verdi command support Tab completion. Try right now to type `verdi` in a terminal and tap Tab twice to have a list of subcommands. Whenever you need the explanation of a command type `verdi help` or add -h flag if you are using any of the `verdi` subcommands. Finally, fields enclosed in angular brackets, such as `<pk>`, are placeholders to be replaced by the actual value of that field (an integer, a string, etc...).

1.1 The list of calculations

Let us try our first `verdi` commands. Type in the terminal

```
verdi calculation list
```

(Note: the first time you run this command, it might take some seconds as it is the first time you are accessing the database in the virtual machine. Following calls will be faster). This will print the list of ongoing calculations, which should be empty. The first output line should look like

```
# Last daemon state_updater check: 0h:00m:18s ago (at 17:17:26 on 2016-05-31)
```

In order to print a list with all calculations that finished correctly in the AiiDA database, you can use the `-s/--states` flag as follows:

```
verdi calculation list --states FINISHED
```

Another very typical option combination allows to get calculations in *any* state (flag `-a`) generated in the past `NUM` days (`-p <NUM>`): e.g., for calculation in the past 1 day: `verdi calculation list -p1 -a`.

Each row of the output identifies a calculation and shows several information about it. For a more detailed list of properties, choose one row by noting down its pk number and type in the terminal

```
verdi calculation show <pk>
```

The output depends on the specific pk chosen and should inform you about the input nodes (e.g. pseudopotentials, kpoints, initial structure, etc.), the output nodes (e.g. output structure, output parameters, etc.). For instance, if you choose `pk=4079`, which identifies a relax of the Si unit cell obtained using Quantum Espresso `pw.x`, the output should look like

```
-----------  ------------------------------------
type         PwCalculation
pk           4079
uuid         ce81c420-7751-48f6-af8e-eb7c6a30cec3
label
description
ctime        2014-10-27 17:51:21.781045+00:00
mtime        2015-10-28 17:58:41.082289+00:00
computer     [1] daint
code         pw-SVN-piz-daint
-----------  ------------------------------------
Using code: pw-SVN-piz-daint
##### INPUTS:
Link label      PK  Type
------------  ----  -------------
pseudo_O      4187  UpfData
pseudo_Ti     4160  UpfData
parameters    4080  ParameterData
settings      3736  ParameterData
kpoints       4370  KpointsData
pseudo_Ba      807  UpfData
structure      285  StructureData
##### OUTPUTS:
Link label                 PK  Type
----------------------  ----  -------------
output_parameters        3516  ParameterData
output_structure         3512  StructureData
retrieved                3514  FolderData
remote_folder            1817  RemoteData
output_trajectory_array  3515  ArrayData
output_kpoints           3511  KpointsData
```

1.2 A typical AiiDA graph

AiiDA stores in the database the inputs required by a calculation as well as the its outputs. These objects are connected in a graph that looks like Fig. 1. We suggest that you have a look to the figure before going ahead.

You can create a similar graph for any calculation node by using the utility `verdi graph show <pk>`. For example, before you obtained information (in text form) for `pk=4079`. To visualize similar information in graph(ical) form, run (replacing `<pk>` with 4079):

```
verdi graph generate <pk>
```

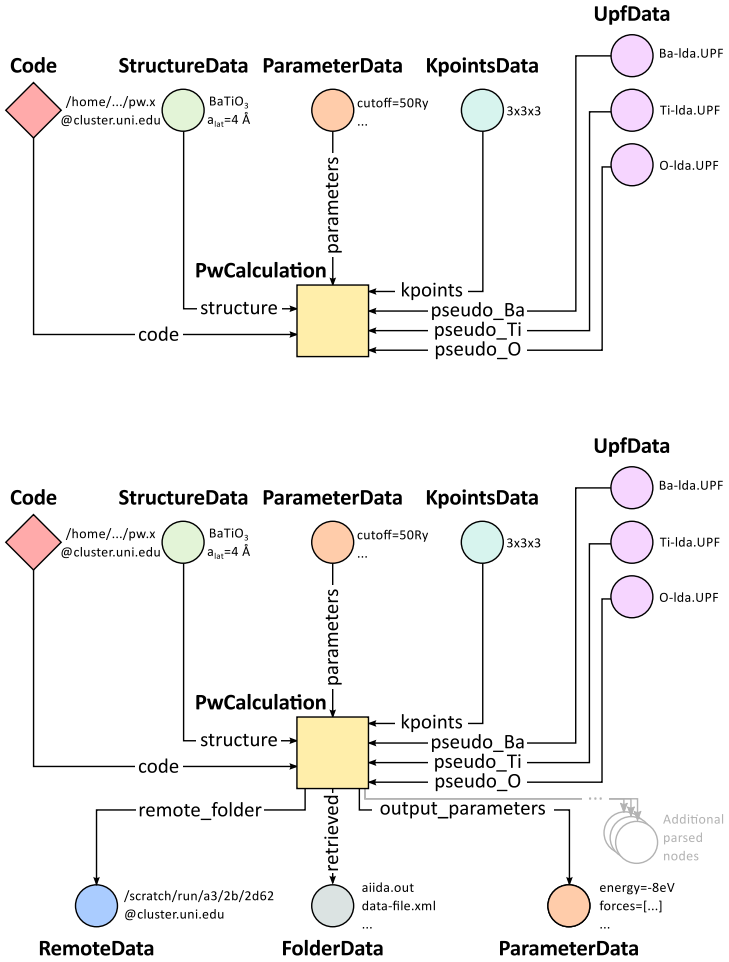

Figure 1 – **(a, top)** Graph with all inputs (data, circles; and code, diamond) to the Quantum Espresso calculation (square) that you will create in Sec. 3 of this tutorial. **(b, bottom)** Same as (a), but also with the outputs that the daemon will create and connect automatically. The RemoteData node is created during submission and can be thought as a symbolic link to the remote folder in which the calculation runs on the cluster. The other nodes are created when the calculation has finished, after retrieval and parsing. The node with linkname "retrieved" contains the raw output files stored in the AiiDA repository; all other nodes are added by the parser. Additional nodes (symbolized in gray) can be added by the parser (e.g., an output StructureData if you performed a relaxation calculation, a TrajectoryData for molecular dynamics, ...).

This command will create the file <pk>.dot that can be rendered by means of the utility `dot`. If you now type

```
dot -Tpdf -o <pk>.pdf <pk>.dot
```

you will create a pdf file `<pk>.pdf`. You can open this file on the Amazon machine by using `evince` or, if you feel that the ssh connection is too slow, copy it via `scp` to your local machine. To do so, if you are using linux/mac, you can type in your *local* machine:

```
scp aiidatutorial:<path_with_the_graph_pdf> <local_folder>
```

and then open the file. Alternatively, you can use graphical software to achieve the same, for instance: WinSCP on Windows, Cyberduck on the Mac, or the "Connect to server" option in the main menu after clicking on the desktop for Ubuntu.

Spend some time to familiarize with the graph structure. Individuate the root node (highlighted in blue) and trace back the parent calculation which produced the structure used as an input. This is an example of a Quantum ESPRESSO pw.x calculation, where the input structure was actually obtained as the output of a previous calculation. We will now inspect the different elements of this graph.

1.3 Inspecting the nodes of a graph

ParameterData and Calculations

Now, let us have a closer look to the some of the nodes appearing in the graph. Choose the node of the type `ParameterData` with input link name `parameters` (ex. pk=4080) and type in the terminal:

```
verdi data parameter show <pk>
```

A `ParameterData` contains a dictionary (i.e., key–value pairs), stored in the database in a format ready to be queried (we will learn how to run queries later on in this tutorial). The command above will print the content dictionary, containing the parameters used to define the input file for the calculation. You can compare the dictionary with the content of the raw input file to Quantum ESPRESSO (that was generated by AiiDA) via the command

```
verdi calculation inputcat <pk>
```

where you substitute the pk of the calculation node (e.g. pk=4079). Check the consistency of the parameters written in the input file and those stored in the ParameterData node. Even if you don't know the meaning of the input flags of a Quantum ESPRESSO calculation, you should be able to see how the input dictionary has been converted to Fortran namelists.

The previous command just printed the content of the "default" input file `aiida.in`. To see a list of all the files used to run a calculation (input file, submission script, etc.) type instead

```
verdi calculation inputls <pk>
```

(Adding a `--color` flag allows you to easily distinguish files from folders by a different coloring).

Once you know the name of the file you want to visualize, you can call the `verdi calculation inputcat` command specifying the path. For instance, to see the submission script, you can do:

```
verdi calculation inputcat <pk> -p _aiidasubmit.sh
```

StructureData

Now let us focus on StructureData objects, such as node pk=285 of the graph. A StructureData object contains a crystal structure. Such objects can be inspected interactively by means of an atomic viewer such as the one provided by `ase`. AiiDA however supports several other viewers such as `xcrysden, jmol, and vmd`. Type in the terminal

```
verdi data structure show --format ase <pk>
```

to show the selected structure. Alternatively, especially if showing them interactively is too slow over SSH, you can export the content of a structure node in various popular formats such as `xyz` or `xsf`. This is achieved by typing in the terminal

```
verdi data structure export --format xsf <pk>  >  <pk>.xsf
```

You can then copy `<pk>.xsf` from the Amazon machine to your local one and then visualize it, e.g. with xcrysden:

```
xcrysden --xsf <pk>.xsf
```

Codes and computers

Let us focus now on the nodes of type `code`. A code represents (in the database) the actual executable used to run the calculation. Find the pk of such a node in the graph and type

```
verdi code show <pk>
```

The command prints information on the plugin used to interface the code to AiiDA, the remote machine on which the code is executed, the path of its executable, etc. To have a list of all available codes type

```
verdi code list
```

Among the entries of the output you should also find the code just shown.

Similarly, the list of computers on which AiiDA can submit calculations is accessible by means of the command

```
verdi computer list -a
```

(`-a` shows all computers, also the one imported in your database but that you did not configure, i.e., to which you don't have access). Details about each computer can be obtained by the command

```
verdi computer show <COMPUTERNAME>
```

Now you have the tools to answer the question:

> What is the scheduler installed on the computer where the calculations of the graph have run?

Calculation results

The results of a calculation can be accessed directly from the calculation node. Type in the terminal

```
verdi calculation res <pk>
```

which will print the output dictionary of the "scalar" results parsed by AiiDA at the end of the calculation. Note that this is actually a shortcut for

```
verdi data parameter show <pk2>
```

where `pk2` refers to the ParameterData node attached as an output of the calculation node, with link name `output_parameters`.

> By looking at the output of the command, what is the Fermi energy of the calculation having pk=4079?

Similarly to what you did for the calculation inputs, you can access the output files via the commands

```
verdi calculation outputls <pk>
```

and

```
verdi calculation outputcat <pk>
```

Use the latter to verify that the Fermi energy that you have found in the last step has been extracted correctly from the output file (Hint: filter the lines containing the string "Fermi", e.g. using `grep`, to isolate the relevant lines).

The results of a calculations are stored in two ways: ParameterData objects are stored in the database, which makes querying them very convenient, whereas ArrayData objects are stored on the disk. Once more, use the command `verdi data array show <pk>` to know the Fermi energy obtained from calculation pk=4079. As you might have realized the difference now is that the whole series of values of the Fermi energy calculated after each relax/vc-relax step are stored.

(Optional section) Comments

AiiDA offers the possibility to attach comments to a calculation node, in order to be able to remember more easily its details. Node pk=4079 has no comment already defined, but you can add a very instructive one by typing in the terminal

```
verdi comment add -c "vc-relax of a BaTiO3 done with QE pw.x" 4079
```

Now, if you ask for a list of all comments associated to that calculation by typing

```
verdi comment show 4079
```

the comment that you just added will appear together with some useful information such as its creator and creation date. We let you play with the other options of `verdi comment` command to learn how to update or remove comments.

1.4 AiiDA groups of calculations

Under AiiDA, calculations can be organized in groups, which are particularly useful to assign a set of calculations to a common project. This allows you to have quick access to a whole set of calculations with no need for tedious browsing of the database or writing complex scripts for retrieving the desired nodes. Type in the terminal

```
verdi group list
```

to have a list of the groups already existing in the database. Choose the pk of the group whose name is `tutorial_pbesol` and look at the calculations that it contains by typing

```
verdi group show <pk>
```

In this case, we have used the name of the group to class calculations according to the pseudopotential that has been used to perform them. Among the rows printed by the last command you will be able to find calculation pk=4079. However, If you want to know the group (or groups) to which a specific node owns, the most practical way is to use

```
verdi group list --node <pk>
```

2 Using the verdi shell and familiarizing with AiiDA objects

In this section we will use an interactive ipython environment with all the basic AiiDA classes already loaded. We propose you two realizations of such a tool. The first consist in a special ipython shell where all the AiiDA classes, methods and functions are accessible. Type in the terminal

```
verdi shell
```

For all the everyday AiiDA-based operations, i.e. creating, querying and using AiiDA objects, the `verdi shell` is probably the best tool. In this case, we suggest that you use two terminals, one for the `verdi shell` and one to execute bash commands.

The second option is based on `jupyter` notebooks and is probably most suitable to the purposes of our tutorial. Go to the browser where you have opened `jupyter` and click `New` → `Python 2` (top right corner). This will open an ipython notebook based on cells where you can type portions of python code. The code will not be executed until you press `Shift+Enter` from within a cell. Type in the first cell

```
%aiida
```

and execute it. This will set exactly the same environment as the `verdi shell`. The notebook will be automatically saved upon any modification and when you think you are done, you can export your notebook in many formats by going to `File` → `Download as`. We suggest you to have a look to the drop-down menus `Insert` and `Cell` where you will find the main commands to manage the cells of your notebook. **The `verdi shell` and the `jupyter notebook` are completely equivalent. Use either according to your personal convenience.**

Note: you will still need sometimes to type command-line instructions in `bash` in the first terminal you opened today. To differentiate these from the commands to be typed in the `verdi shell`, the latter will be marked in this document by a vertical line on the left, like:

```
some verdi shell command
```

while command-line instructions in `bash` to be typed on a terminal will be encapsulated between horizontal lines:

```
some bash command
```

Alternatively, to avoid changing terminal, you can execute `bash` commands within the `verdi shell` or the notebook adding an exclamation mark before the command itself

```
!some bash command
```

2.1 Loading a node

Most AiiDA objects are represented by nodes, identified in the database by its pk number (an integer). You can access a node using the following command in the shell:

```
node = load_node(PK)
```

Load a node using one of the calculation pks visible in the graph you displayed in the previous section of the tutorial. Then get the energy of the calculation with the command

```
node.res.energy
```

You can also type

```
node.res.
```

and then press `TAB` to see all the possible output results of the calculation.

2.2 Loading different kinds of nodes

2.2.1 Pseudopotentials

From the graph displayed in Section 1.2, find the pk of the barium pseudopotential file (LDA). Load it and verify that it describes barium. Type

```
upf = load_node(PK)
upf.element
```

All methods of `UpfData` are accessible by typing `upf.` and then pressing `TAB`.

2.2.2 k-points

A Set of k-points in the Brillouin zone is represented by an instance of the `KpointsData` class. Choose one from the graph of Section 1.2, load it as `kpoints` and inspect its content:

```
kpoints.get_kpoints_mesh()
```

Then get the full (explicit) list of k-points belonging to this mesh using

```
kpoints.get_kpoints_mesh(print_list=True)
```

If you incurred in a `AttributeError`, it means that the kpoints instance does not represent a regular mesh but rather a list of k-points defined by their crystal coordinates (typically used when plotting a band structure). In this case, get the list of k-points coordinates using

```
kpoints.get_kpoints()
```

If you prefer Cartesian (rather than crystal) coordinates, type

```
kpoints.get_kpoints(cartesian=True)
```

For later use in this tutorial, let us try now to create a kpoints instance, to describe a regular $2 \times 2 \times 2$ mesh of k-points, centered at the Gamma point (i.e. without offset). This can be done with the following commands:

```
from aiida.orm.data.array.kpoints import KpointsData
kpoints = KpointsData()
kpoints_mesh = 2
kpoints.set_kpoints_mesh([kpoints_mesh,kpoints_mesh,kpoints_mesh])
kpoints.store()
```

The import performed in the first line is however unpractical as it requires to remember the exact location of the module containing the KpointsData class. Instead, it is easier to use the `DataFactory` function instead of an explicit import.

```
KpointsData = DataFactory("array.kpoints")
```

This function loads the appropriate class defined in a string (here `array.kpoints`).[1] Therefore, `KpointsData` is not a class instance, but the kpoints class itself!

2.2.3 Parameters

Nested dictionaries with individual parameters, as well as lists and arrays, are represented in AiiDA with `ParameterData` objects. Get the PK and load the input parameters of a calculation in the graph of Section 1.2. Then display its content by typing

```
params.get_dict()
```

where `params` is the `ParameterData` node you loaded. Modify the dictionary content so that the wave-function cutoff is now set to 20 Ry. Note that you cannot modify an object already stored in the database. To save the modification, you must create a new ParameterData object. Similarly to what discussed before, first load the `ParameterData` class by typing

```
ParameterData = DataFactory('parameter')
```

Then an instance of the class (i.e. the parameter object that we want to create) is created and initialized by the command

```
new_params = ParameterData(dict=YOUR_DICT)
```

where `YOUR_DICT` is the modified dictionary. Note that the parameter object is not yet stored in the database. In fact, if you simply type `new_params` in the verdi shell, you will be prompted with a string notifying you the "unstored" status. To save an entry in the database corresponding to the `new_params` object, you need to type a last command in the verdi shell:

```
new_params.store()
```

[1] The string provided to the `DataFactory` encodes both the location and the name of the required class according to some specific rules.

2.2.4 Structures

Find a structure in the graph of Section 1.2 and load it. Display its chemical formula, atomic positions and species using

```
structure.get_formula()
structure.sites
```

where `structure` is the structure you loaded. If you are familiar with ASE and PYMATGEN, you can convert this structure to those formats by typing

```
structure.get_ase()
structure.get_pymatgen()
```

Let's try now to define a new structure to study, specifically a silicon crystal. In the `verdi shell`, define a cubic unit cell as a 3×3 matrix, with lattice parameter $a_{lat} = 5.4$ Å:

```
alat = 5.4
the_cell = [[alat/2,alat/2,0.],[alat/2,0.,alat/2],[0.,alat/2,alat/2]]
```

(default units are in Å) Structures in AiiDA are instances of `StructureData` class: load it in the verdi shell

```
StructureData = DataFactory("structure")
```

Now, initialize the class instance (i.e. is the structure we want to study) by the command

```
structure = StructureData(cell=the_cell)
```

which sets the cubic cell defined before. From now on, you can access the cell with the command

```
structure.cell
```

Finally, append each of the 2 atoms of the cell command. You can do it using commands like

```
structure.append_atom(position=(alat/4.,alat/4.,alat/4.),symbols="Si")
```

for the first 'Si' atom. Repeat it for the other atomic site $(0, 0, 0)$. You can access and inspect[2] the structure sites with the command

```
structure.sites
```

If you make a mistake, start over from `structure = StructureData(cell=the_cell)`, or equivalently use `structure.clear_kinds()` to remove all kinds (atomic species) and sites. Alternatively, AiiDA structures can also be converted directly from ASE [2] structures using[3]

```
from ase.lattice.spacegroup import crystal
ase_structure = crystal('Si', [(0,0,0)], spacegroup=227,
                cellpar=[alat, alat, alat, 90, 90, 90],primitive_cell=True)
structure=StructureData(ase=ase_structure)
```

Now you can store the new structure object in the database with the command:

```
structure.store()
```

Finally, we can also import the silicon structure from an external (online) repository such as the Crystallography Open Database [6]:

```
from aiida.tools.dbimporters.plugins.cod import CodDbImporter
importer = CodDbImporter()
for entry in importer.query(formula='Si',spacegroup='F d -3 m'):
        structure = StructureData()
        structure.set_ase(entry.get_ase_structure())
        print "Formula", structure.get_formula()
        print "Unit cell volume: ", structure.get_cell_volume()
```

In that case two duplicate structures are found for Si.

[2] if you set the structure incorrectly, for example with overlapping atoms, it is very likely that any DFT code will fail!

[3] We purposefully do not provide advanced commands for crystal structure manipulation, because python packages that accomplish such tasks already exist (such as ASE or pymatgen).

2.3 Accessing inputs and outputs

Load again the calculation node used in Section 2.1:

```
calc = load_node(PK)
```

Then type

```
calc.inp.
```

and press `TAB`: you will see all the link names between the calculation and its input nodes. You can use a specific linkname to access the corresponding input node, e.g.:

```
calc.inp.structure
```

You can use the `inp` method multiple times in order to browse the graph. For instance, if the input structure node that you just accessed is the output of another calculation, you can access the latter by typing

```
calc2 = calc.inp.structure.inp.output_structure
```

Here `calc2` is the `PwCalculation` that produced the structure used as an input for `calc`.

Similarly, if you type:

```
calc2.out.
```

and then `TAB`, you will list all output link names of the calculation. One of them leads to the structure that was the input of `calc` we loaded previously:

```
calc2.out.output_structure
```

Note that links have a single name, that was assigned by the calculation that used the corresponding input or produced the corresponding output, as illustrated in Fig. 1.

For a more programmatic approach, you can get a list of the inputs and outputs of a node, say `calc`, with the methods

```
calc.get_inputs()
calc.get_outputs()
```

Alternatively, you can get a dictionary where the keys are the link names and the values are the linked objects, with the methods

```
calc.get_inputs_dict()
calc.get_outputs_dict()
```

Note: You will sometime see entries in the dictionary with names like `output_kpoints_3511`. These exist because standard python dictionaries require unique key names while link labels may not be unique. Therefore, we use the link label plus the PK separated by underscores.

2.4 Pseudopotential families

Pseudopotentials in AiiDA are grouped in "families" that contain one single pseudo per element. We will see here how to work with UPF pseudopotentials (the format used by Quantum ESPRESSO and some other codes). Download and untar the SSSP [5] pseudopotentials via the commands:

```
wget http://www.materialscloud.ch/sssp/pseudos/SSSP_eff_PBESOL.tar.gz
tar -zxvf SSSP_eff_PBESOL.tar.gz
```

Then you can upload the whole set of pseudopotentials to AiiDA by to the following `verdi` command:

```
verdi data upf uploadfamily SSSP_eff_PBESOL 'SSSP' 'SSSP pseudopotential
library'
```

In the command above, `SSSP_eff_PBESOL` is the folder containing the pseudopotentials, 'SSSP' is the name given to the family and the last argument is its description.
Finally, you can list all the pseudo families present in the database with

```
verdi data upf listfamilies
```

3 Submit, monitor and debug calculations

The goal of this section is to understand how to create new data in AiiDA. We will launch a total energy calculation and check its results. We will introduce intentionally some common mistakes along the process of defining and submitting a calculation and we will explain you how to recognize and correct them. While this debugging is done here 'manually', workflows (that we will learn later in this tutorial) can automate this procedure considerably. For computing the DFT energy of the silicon crystal (with a PBE functional) we will use Quantum ESPRESSO [1] (v. 5.1), in particular the code PWscf. AiiDA already provides a set of "quantumespresso" plugins; in particular the EPFL version extends the MIT version and covers most functionalities.

3.1 The AiiDA daemon

First of all check that the AiiDA daemon is actually running. The AiiDA daemon is a program running all the time in the background, checking if new calculations appear and need to be submitted to the scheduler[1]. The daemon also takes care of all the necessary operations before the calculation submission[2], and after the calculation has completed on the cluster.[3] Type in the terminal

```
verdi daemon status
```

If the daemon is running, the output should look like

```
# Most recent daemon timestamp:0h:00m:26s ago
## Found 1 process running:
   * aiida-daemon[aiida-daemon] RUNNING    pid 15044, uptime 3 days, 15:38:41
```

If this is not the case, type in the terminal

```
verdi daemon start
```

to start the daemon.

3.2 Creating a new calculation

To launch a calculation, you will need to interact with AiiDA mainly in the `verdi shell`. We strongly suggest you to first try the commands in the shell, and then copy them in a script "test_pw.py" using a text editor. This will be very useful for later execution of a similar series of commands.

The best way to run python scripts using AiiDA functionalities is to run them in a terminal by means of the command

```
verdi run <scriptname>
```

Every calculation sent to a cluster is linked to a code, which describes the executable file to be used. Therefore, first load the suitable code:

```
code = Code.get_from_string(<codename>)
```

Here `Code` is the general AiiDA class handling all possible codes, and `code` is a class instance tagged as `<codename>` (see the first part of the tutorial for listing all codes installed in your AiiDA machine). For this example use codename `'pw-5.1@localhost'`.

AiiDA calculations are instances of the class `Calculation`, more precisely of one of its subclasses, each corresponding to a code specific plugin (for example, the PWscf plugin). We create a new calculation using the `new_calc` method of the `code` object:

```
calc = code.new_calc()
```

[1] i.e. first queued on the cluster, and then run when the queuing system allows it

[2] creating a directory where to run and copying there the input files

[3] retrieving and copying some of the output files in the database repository, parsing one or several output files, attaching some data to the calculation and storing them in the database.

This creates and initializes an instance of the `PwCalculation` class, the subclass associated with the `quantumespresso.pw` plugin. Sometimes, you might find convenient to annotate information assigning a (short) label or a (long) description, like:

```
calc.label="PW test"
calc.description="My first AiiDA calc with Quantum ESPRESSO on BaTiO3"
```

This information will be saved in the database for later query or inspection.

Now you have to specify the number of machines (a.k.a. cluster nodes) you are going to run on and the maximum time allowed for the calculation — this information is passed to the scheduler that handles the queue:

```
calc.set_resources({"num_machines": 1})
calc.set_max_wallclock_seconds(30*60)
```

3.2.1 Preparation of inputs

Quantum ESPRESSO requires an input file containing Fortran namelists and variables, plus some cards sections (the documentation is available online[4]). The Quantum ESPRESSO plugin of AiiDA requires quite a few nodes in input, which are documented online[5]. Here we will instruct our calculation with a minimal configuration for computing the energy of silicon. We need:

1. Pseudopotentials
2. a structure
3. the k-points
4. the input parameters

We leave the parameters as the last thing to setup and start with structure, k-points, and pseudopotentials.

Use what you learned in the previous section and define these two kinds of objects in this script. Define in particular a silicon structure and a mesh 2×2×2 of k-points. Notice that if you just copy and paste the code that you executed previously, you will create duplicated information in the database (i.e. every time you will execute the script, you will create another StructureData, another KpointsData, ...). In fact, you already have the opportunity to re-use an already existing structure.[6] Use therefore a combination of the bash command `verdi data structure list` and of the shell command `load_node()` to get an object representing the structure created earlier.

3.2.2 Attaching the input information to the calculation

So far we have defined (or loaded) some of the input data, but we haven't instructed the calculation to use them. This is done with the `calc.use_<linkname>` commands:

```
calc.use_structure(structure)
calc.use_kpoints(kpoints)
```

Moreover, PWscf needs also information on the pseudopotentials, specified by UpfData objects. To simplify the task of choosing pseudopotentials, use the Upf families that you created before. You can list the preconfigured families from the command line:

```
verdi data upf listfamilies
```

Pick the one you configured earlier (e.g., 'SSSP') and link it to the calculation using the command[7]:

```
calc.use_pseudos_from_family('SSSP')
```

[4] http://www.quantum-espresso.org/wp-content/uploads/Doc/INPUT_PW.html

[5] http://aiida-core.readthedocs.io/en/latest/plugins/quantumespresso/pw.html

[6] However, to avoid duplication of KpointsData, you should first learn how to query the database, therefore we will ignore this duplication issue for now.

[7] This should be the most convenient way of setting pseudopotentials in most cases. Although we will not cover it in this tutorial, you can have more control and decide manually which UpfData you want to set for each element (check the plugin documentation in the manual).

3.2.3 Preparing and debugging input parameters

The last thing we miss is a set of parameters (i.e. cutoffs, convergence thresholds, etc...) to launch the Quantum ESPRESSO calculation. This part requires acquaintance with Quantum ESPRESSO and, very often, this is the part to tune when a calculation shows a problem. Let's therefore use this part of the tutorial to learn how to debug problems, and **let's introduce errors intentionally**. Note also that some of the problems we will investigate appear the first times you launch calculations and can be systematically avoided by using workflows.

Let's define a set of input parameters for Quantum ESPRESSO, preparing a dictionary of the form:

```
parameters_dict = {'CONTROL': {'calculation': 'scf',
                               'tstress': True,
                               'tprnfor': True,
                               },
                   'SYSTEM': {'ecutwfc': 30.,
                              'ecutrho': 200.,
                              'mickeymouse': 240.,
                              },
                   'ELECTRONS': {'conv_thr': 1.e-8,
                                 },
                   }
```

This dictionary is almost a valid input for the Quantum ESPRESSO plugin, except for an invalid key called "mickeymouse". When Quantum ESPRESSO receives an unrecognized key (even when you misspell one) its behavior is to stop almost immediately. By default, the AiiDA plugin will not validate your input and simply pass it over. Therefore let's pass this dictionary to the calculation and observe this unsuccessful behavior.

As done before, load the ParameterData class

```
ParameterData = DataFactory("parameter")
```

and create an instance of the class containing all the input parameters you just defined

```
parameters = ParameterData(dict=parameters_dict)
```

Finally, link it to the calculation:

```
calc.use_parameters(parameters)
```

3.2.4 Make test

At this stage, you have recreated in memory (it's not yet stored in the database) the input of the graph shown in Fig. 1a, whereas the outputs will be created later by the daemon.

In order to check how AiiDA creates the actual input files for the calculation, we can simulate the submission process with the (otherwise optional) command

```
calc.submit_test()
```

This creates a folder of the form `submit_test/[date]-0000[x]` in the current directory. Check (in your second terminal) the input file `aiida.in` within this folder, comparing it with the content of the input data nodes you created earlier, and that the 'pseudo' folder contains the needed pseudopotentials. You can also check the submission script `_aiidasubmit.sh` (the scheduler that is installed on the machine is PBS, so AiiDA creates the files with the proper format for this scheduler). Note: you cannot correct the input file from the "submit_test" folder: you have to correct the script and re-execute it; the files created by `submit_test()` are only for final inspection.

3.2.5 Storing and submitting the calculation

Up to now the calculation `calc` is kept in memory and not in the database. Just before submission, we store it (and all its inputs parameters, k-points, structure) in the database:

```
calc.store_all()
```

This also associates a 'pk' (an integer ID) to the calculation (typing `calc.pk` will print this number). You can also attach any additional attributes of your choice, which are called "extra" and defined in as key-value pairs. For example, you can add an extra attribute called 'element', with value 'Si' through

```
calc.set_extra("element","Si")
```

You will see later the advantage of doing so for querying.

Finally, submit the job with

```
calc.submit()
```

The last command simply changes the state of the calculation from NEW to TOSUBMIT. Every few seconds, the daemon scans for calculations in the TOSUBMIT state and performs all the operations to do the actual submission: therefore, it will take a few seconds before the calculation is actually submitted.

N.B. If the daemon is not running the calculation will hang in the TOSUBMIT state!

3.2.6 Checking the status of the calculation

You can check the calculation status from the command line:

```
verdi calculation list
```

Note that `verdi` commands can be slow in this tutorial when the calculation is running (because you just have one CPU which is also used by the PWscf calculation). By default, the command only prints calculations that are still being handled by the daemon, i.e. those neither in the FINISHED state nor in one of the possible failure states (FAILED, PARSINGFAILED, SUBMISSIONFAILED, RETRIEVALFAILED). To see calculations that have finished, use instead

```
verdi calculation list -a -p1
```

as explained in the first section. While the calculation is running, type

```
verdi calculation inputls <pk_number> -c
```

with `pk_number` the pk number of your calculation. This will show the contents of the input directory (`-c` prints directories in colour). Then you can also check the content of the actual input file with

```
verdi calculation inputcat <pk_number> | less
```

3.3 Troubleshooting

After all this work the calculation should end up in a FAILED state, as expected since we used an invalid key in the input parameters. Since situations like this happens in real life, we built in AiiDA the tools to traceback the problem source and correct it.

A first way to proceed is the manual inspection of the output file of PWscf. You can visualize it with:

```
verdi calculation outputcat <pk_number> | less
```

This can be a good primer for problem inspection. For something more compact, you can also try to inspect the calculation log (from AiiDA):

```
verdi calculation logshow <pk_number>
```

If the calculation has incurred in some mistake, this log shows a handful warnings coming from various processes, such as the daemon, the parser of the output or the scheduler on the cluster. In production runs, errors will mostly come from an unexpected termination of the PWscf calculation. The most programmatic way to handle these errors is to inspect the warnings key is to load a calculation object, say `calc`, and the use the following method:

```
calc.res.warnings
```

This will prints a list of strings reporting errors experienced during the execution, that can be easily read in python (and thus addressed programmatically), but are also reported in the calculation log. With any of these three methods you can understand that the problem is something like an 'invalid input key', which is exactly what we did.

Let's use a parameters dictionary that actually works. Modify the script `test_pw.py` script modifying the parameter dictionary as

```
parameters_dict = {
    "CONTROL": {"calculation": "scf",
                },
    "SYSTEM": {"ecutwfc": 30.,
               "ecutrho": 200.,
               },
    "ELECTRONS": {"conv_thr": 1.e-6,
                  }
    }
```

If you launch the modified script by typing

```
verdi run test_pw.py
```

you should now be able to see a calculation reaching successfully the FINISHED state. Now you can access the results as you have seen earlier. For example, note down the pk of the calculation so that you can load it in the `verdi shell` and check the total energy with the commands:

```
calc=load_node(<pk>)
calc.res.energy
```

4 Queries in AiiDA: The QueryBuilder

In this part of the tutorial we will focus on how to query our database using a querying tool for AiiDA called the *QueryBuilder*. Queries are, very loosely defined, questions to your database. We will first show you some simple examples and tasks on how to explore your database. Then we will proceed to a more concrete exercise on the screening of magnetic and metallic perovskites.

Task 1 - Introduction to QueryBuilder

In this task we will use the QueryBuilder to do some basic queries and understand our database. As a first step we should import our querying tool, the *QueryBuilder*.

```
from aiida.orm.querybuilder import QueryBuilder
```

After the above import, we create our first query. To do so, we will have to instantiate a QueryBuilder instance:

```
qb = QueryBuilder()
```

Our query is still empty, we have not yet defined what we want to see. For example, we will ask for all the nodes of our database. This is as simple as appending the Node class to the query that we construct.

Node & subclasses	Number in DB
Node	4707
StructureData	621
ParameterData	1338
KpointsData	861
UpfData	99
JobCalculation	448

Table 1: List of some Node subclasses and how many times they occur in our test database.

```
qb.append(Node)
```

At this point, we can finish our query by asking back all nodes and by typing

```
qb.all()        # Returns all nodes in the database
```

However, this command will return us all the Nodes directly, which may not be the most wise thing to do considering that is the biggest family of AiiDA stored objects that we can query. To understand the size of the result, we can type the following command:

```
qb.count()      # Returns an integer, the number of nodes in the database
```

If you are interested to retrieve a subclass of a node, append that specific subclass instead of Node:

```
StructureData = DataFactory('structure')
qb = QueryBuilder()          # Creating a new QueryBuilder instance
qb.append(StructureData)     # Telling the QueryBuilder instance that I want structures
qb.all()                     # Asking for all the results!
```

Exercise:

- Try now to find the number of instances for some subclasses of Node (e.g. StructureData, ParameterData, etc.) that are stored in your database. The result should look like Table 1. Of course, the numbers can be different!

Comment: If you are familiar with the SQL (Structured Query Language) syntax then you may wonder what the issued SQL command is. This can be easily seen by typing:

```
str(qb)
```

Comment: If you want to get inspired by the available QueryBuilder options you can just press the *tab* key in an interactive shell (after typing `qb.`) to see the available options.

Comment: After you run a query, a new `QueryBuilder` instance needs to be defined if you want to make a new query.

Task 2 - Projections and filters

Operator	Datatype	Example
==	All	{'==':12}
in	All	{'in':['FINISHED', 'PARSING']}
>, <, <=, >=	floats, integers, dates	{'>':5.2}
like	Chars	{'like':'calculation%'}
ilike	Chars	{'ilike':'caLculAtioN%'}
or		{'or':[{'<':5.3}, {'>':6.3}]}
and		{'and':[{'>':5.3}, {'<':6.3}]}

Table 2: Operators currently implemented for all backends.

In database language performing a projection means to extract one or more specific columns from a table. In the AiiDA language this is equivalent to say that we select what properties a query should return out of the queried objects. For example, we might be interested only in the id of a set of nodes (or their creation date, or any stored value). To this purpose we should suitably instruct a QueryBuilder object by means of the "project" key. For example, if we would like to get all the ids of the nodes, we would type the following:

```
qb = QueryBuilder()
qb.append(Node, project=["id"])
qb.all()
```

Please note that if you would like to perform an operation on the *pk* of a node, you should use the keyword *id* in QueryBuilder queries.

Most likely, performing a query implies to select only those elements that fulfill certain criteria. For example, we might want to select all the calculations that were launched on a specific date. In database language, this is called "adding a filter" to a query. A filter is a boolean operator that returns True or False. Table 2 lists all operators that we implemented. A selection of entities and some of their properties that you can use at your projections and filters can be found at table Table 3.

If you want to add filters to your query, you simply add the *filters* keyword with a dictionary. Suppose you want to know the creation date of a structure of which you know the uuid:

```
qb = QueryBuilder()             # Instantiating a new QueryBuilder
qb.append(
    StructureData,              # I want structures!
    project=["ctime"],          # I'm interested in creation time!
    filters={"uuid": {
            "==":"ace6523a-2019-47c4-98a3-48429265f62c"
        }})                     # I want the structure with this UUID
qb.all()
```

Try it out! There is also the possibility to combine multiple filters on the same object using the "and" or the "or" keyword in the filter section. Let's see an example.

```
from datetime import datetime, timedelta
qb = QueryBuilder()
qb.append(
    StructureData,
    project=["uuid"],           #  I want to see only the UUID
    filters={"or":[             # First filter is an or statement
            {"ctime": {">":datetime.now() - timedelta(days=12)}},
            {"label": "graphene"}
        ]}
   )
qb.all()
```

In the above example we added an "or" keyword between the two filters. The query return every structure in the database that was created in the last 12 days or is named "graphene".

Hints for the exercises:

- The operator '>', '<' works with date-type properties with the expected behavior.
- For your date comparisons you will need to create a `datetime` object to which you can assign a date of your preference. You will have to do the necessary import (`from datetime import datetime`) and create an object by giving a specific date. E.g. datetime(2015, 12, 26). For further information, you can consult the Python's online documentation.

Entity	Properties
Node	id, uuid, type, label, description, ctime, mtime
Computer	id, uuid, name, hostname, description, enabled, transport_type, scheduler_type
User	id, email, first_name, last_name, institution
Group	id, uuid, name, type, time, description

Table 3: A selection of entities and some of their properties.

Entity from	Entity to	Relationship	Explanation
Node	Node	input_of	One node as input of another node
Node	Node	output_of	One node as output of another node
Node	Node	ancestor_of	One node as the ancestor of another node
Node	Node	descendant_of	One node as descendant of another node
Group	Node	group_of	The group of a node
Node	Group	member_of	The node is a member of a group
Computer	Node	computer_of	The computer of a node
Node	Computer	has_computer	The node of a computer
User	Node	creator_of	The creator of a node is a user
Node	User	created_by	The node was created by a user

Table 4: Available relationships

Exercises:

- Write a query that returns all instances of StructureData that have been created after the 1st of January 2016.
- Write a query that returns all instances of Group whose name starts with "tutorial".

Task 3 - Defining relationships

In the previous tasks we saw how to select specific entities from AiiDA, how to apply projections on their properties and how to apply filters. Moreover, you should know by now how to write complex filters including "and" and "or" keywords. In this task we will see how to associate entities by defining relationships among them.

Defining a relationship between two entities is as easy as appending another entity to the QueryBuilder query. For example, let's look at the following query:

```
from aiida.orm import Group, JobCalculation
qb = QueryBuilder()
qb.append(JobCalculation,
        tag="mycalculation",            # I tag, so I can refer to this entity later
        project=["*"]                   # I'm asking for ORM instances ("*")
qb.append(Group,                        # Also asking for groups
        group_of="mycalculation",       # I want the calculation to be part of the group
        filters={                       # A filter, the name has to be "pbe_calculation"
            "name":{"==":"pbe_calculation"}
        })
qb.all()
```

This returns all jobs that belong to a Group with the name "pbe_calculation". The *project=["*"]* returns the AiiDA orm instance (i.e. an instance of *aiida.orm.calculation.job.JobCalculation*). There are few details that we should remember from the above query:

- We *append* to the QueryBuilder instance new entities, and specify how they linked to previous entities with keywords ("input_of", "output_of", "group_of"). The possible relationship keywords can be seen at table 4.
- Therefore, we have to *tag* the entities that we want to reference. We pass these tags along with the keyword, to define a relationship.

Exercise:

- Write a query that returns all the StructureData that are an input of a JobCalculation.

More (optional) exercises on entity relationships can be found at the appendix.

Task 4 - Attributes and extras

Node and its subclasses have properties stored in key/value format. These are called *attributes* and *extras* and can be used in filters and projections as the other properties that we have seen previously.

Let's project the value of the property *attributes.energy_smearing*.

```
qb = QueryBuilder()
qb.append(
        ParameterData,
        project=["attributes.energy_smearing"]
    )
qb.all()
```

The above query takes the attributes properties of every ParameterData and searches for a key called energy_smearing. If that key is found, the value is projected, otherwise the value is None. Try it out!

Hints for the exercises:

- If you are unsure about the key of the node that you would like to project, you can print all the attributes of a specific node to get some inspiration. This can be done by calling the `.get_attrs()` method of a specific node.
- The pseudopotentials are stored in AiiDA in with the help of the ORM class UpfData. The element that the pseudopotential correspond to is stored in the *attributes.element* property.

Exercises:

- Print all the attributes of any ParameterData node stored in your database.
- Write a query that checks if you have the pseudopotentials for the element *Si*. Do the same for *C*.

More (optional) exercises on attributes and extras can be found at the appendix.

Task 5 - A small "high-throughput" analysis

In this part of the tutorial, we will focus on how to systematically retrieve, query and analyze the results of multiple calculations using AiiDA. We know you're able to do this yourself, but to save time, a set of calculations have already been done with AiiDA for you on 57 perovskites, using three different pseudopotential families (LDA, PBE and PBESOL, all from GBRV 1.2 [4]). These calculations are spin-polarized (without spin-orbit coupling), use a Gaussian smearing and perform a variable-cell relaxation of the full unit cell. The idea of this part of the tutorial is to "screen" for magnetic and metallic perovskites in a "high-throughput" way.

As you learned in the first part of the tutorial, AiiDA allows to organize calculations in groups. Once more check the list of groups in your database by typing

```
verdi group list
```

The calculations needed for this task were put in three different groups whose names start with "tutorial_" (one for each pseudopotential family). The main task is to make a plot showing, for all perovskites and for each pseudopotential family, the total magnetization and the $-TS$ contribution from the smearing to the total energy. An example is shown in Figure 2.

Preparing the analysis

Let us now guide you through the definition of the query that allows you to retrieve the relevant data.

Hint for the exercise:

- To select multiple names in a filter your filter-dictionary should look like

```
qb.append(Group, filters={"name": {"in": ["name1","name2","name3"]}})
```

Exercise:

Please perform the following steps

1. Write a query to retrieve the three groups and project the respective names.
2. Extend the query to the PwCalculation nodes that are members of the selected groups.
3. Extend the query so that also the chemical formula of the input structure of each calculation is returned. For simplicity the formulas have been added in the extras of each structure node under the key *"formula"*. The chemical formula of a StructureData node can also be accessed by the method `structure.get_formula()`
4. Every successful PwCalculation has in output a ParameterData instance that stores the results as key-value pairs. You can find these pairs among the attributes. To facilitate querying, the parser takes care of storing values always in the same units, and these are documented. For convenience, the units are also added as key/value pairs (with the same key name, but with `_units` appended). Extend the query so that also the output ParameterData of each calculation is returned. Project only the attributes relevant to your analysis (like smearing energy, ...).

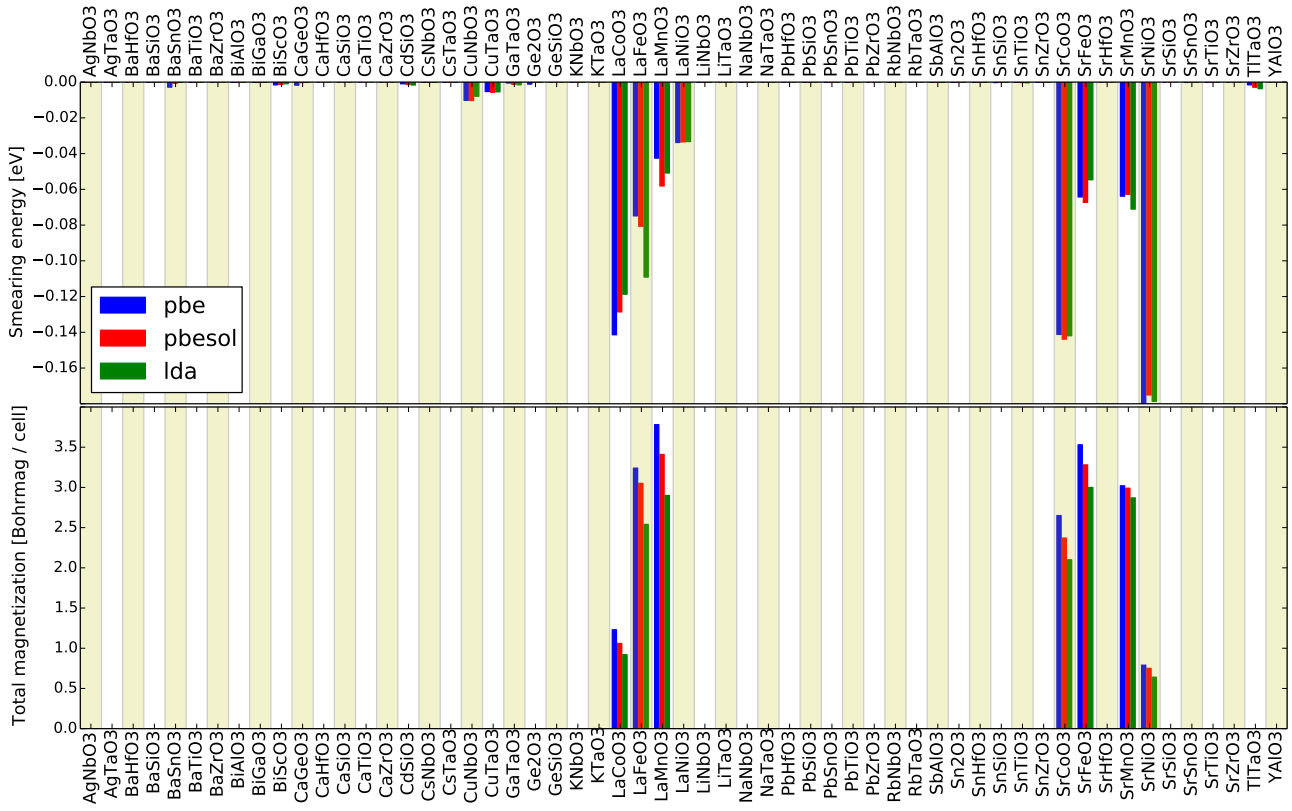

Figure 2 – The contribution from the smearing to the total energy (upper) and the magnetization per unit cell (lower) in all the perovskites analyzed and for three different pseudopotential families.

Running the query and plotting the results

Now that your query is ready just run it:

```
res = qb.dict()
```

The above results returns a list of dictionaries. In order to be able to plot the data that you have retrieved a script called `plot_calculation_results.py` was provided to you. This script reads an input text file containing the data and produces a plot similar to Figure 2.

You should prepare this input text file in the following format, formatting the data present in `res`. Each row in the input file represents one calculation and contains the following informations separated by a comma:

1. The structure formula.
2. The name of the pseudo family. Remove the tutorial_ substring from the name if it bothers you.
3. The smearing energy
4. The unit used for the smearing energy.
5. The magnetization.
6. The unit of the magnetization.

As an example, the first three lines can look like:

```
Sn2O3,  pbesol, -1.95921960844e-05,  eV,  0.0,  Bohrmag / cell
CaSiO3,    pbe,                0.0,  eV,  0.0,  Bohrmag / cell
CuNbO3,    pbe,    -0.010202636111,  eV,  0.0,  Bohrmag / cell
```

If you are producing the desired output, write that output to a file (here we will call it `res.txt`). If you are using *jupyter*, you can plot the files and see the results directly in the browser. First, in a cell, do

```
cd ~/tutorial_scripts/
```

to go to the right folder, and then in the following cell type:

```
%pylab inline
import plot_calculation_results
plot_calculation_results.plot_results('../res.txt')
```

(possibly replacing `../res.txt` with the correct location of the file).

Otherwise, if you are not using *jupyter*, to plot your data just go in the shell and change directory to the subfolder `tutorial_scripts`. Then type in the shell

```
python plot_calculation_results.py ../res.txt
```

(or replace `../res.txt` with the correct location of the `res.txt` file). The `plot_calculation_results.py` file is already provided to you among the scripts given to you for the tutorial, in the `tutorial_scripts` folder. If everything is right, you should get a plot similar to Figure 2. You can also pass an option to store the output as a plot on a file:

```
python plot_calculation_results.py res.txt -o myresult.pdf
```

Exercise:

- Look at the plot that you produced. Which of the perovskites are metals?

5 AiiDA Workflows: workfunctions, workchains

The aim of the last part of this tutorial is to introduce the concept of workflows in AiiDA.

In this section, we will ask you to:

1. Understand how to keep the provenance when running small python scripts to convert one data object into another (postprocessing, preparation of inputs, ...)
2. Understand how to represent simple python functions in the AiiDA database
3. Learn how to write a simple workflow in AiiDA (without and with remote calculation submission)
4. Learn how to write a workflow with checkpoints: this means that, even if your workflows requires external calculations to start, them and their dependences are managed through the daemon. While you are waiting for the calculations to complete, you can stop and even shutdown the computer in which AiiDA is running. When you restart, the workflow will continue from where it was.
5. (optional) Go a bit deeper in the syntax of workflows with checkpoints (WorkChain), e.g. implementing a convergence workflow using `while` loops.

A note: this is probably the most "complex" part of the tutorial. We suggest that you try to understand the underlying logic behind the scripts, without focusing too much on the details of the workflows implementation or the syntax. If you want, you can then focus more on the technicalities in a second reading.

5.1 Introduction

The ultimate aim of this section is to create a workflow to calculate the equation of state of silicon. This is a very common task for an ab-initio researcher. An equation of state consists in calculating the total energy E as a function of the unit cell volume V. The minimal energy is reached at the equilibrium volume $V^\star$. Equivalently, the equilibrium is defined by a vanishing pressure $p = -dE/dV$. In the vicinity of the minimum, the functional form of the equation of state can be approximated by a parabola. Such an approximation greatly simplifies the calculation of the bulk modulus, that is proportional to the second derivative of the energy d^2E/dV^2 (a more advanced treatment requires fitting the curve with, e.g., the Birch–Murnaghan expression).

The process of calculating an equation of state puts together several operations. First, we need to define and store in the AiiDA database the basic structure of, e.g., bulk Si. Next, one has to define several structures with different lattice parameters. Those structures must be connected between them in the database, in order to assure that their provenance is recorded. In other words, we want to be sure that in the future we will know that if we find a bunch of rescaled structures in the database, they all descend from the same one. How to link two nodes in the database in a easy way is the subject of Sec. 5.2.

In the following sections, the newly created structures will then serve as an input for total energy calculations performed, in this tutorial, with Quantum ESPRESSO. This task is very similar to what you have done in the previous part of the tutorial. Finally, you will fit the resulting energies as a function of volume to get the bulk modulus. As the EOS task is very common, we will show how to automate its computation with workflows, and how to deal with both serial and parallel (i.e., independent) execution of multiple tasks. Finally, we will show how to introduce more complex logic in your workflows such as loops and conditional statements (Sec. D), with an example on a convergence loop to find iteratively the minimum of an EOS.

5.2 Workfunctions: a way to generalize provenance in AiiDA

Imagine to have a function that takes as input a string of the name of a chemical element and generates the corresponding bulk structure as a StructureData object. The function might look like this (you will find this function in the folder `/home/aiida/tutorial_scripts/create_rescale.py` on your virtual machine):

```
def create_diamond_fcc(element):
    """
    Workfunction to create the crystal structure of a given element.
    For simplicity, only Si and Ge are valid elements.
    :param element: The element to create the structure with.
```

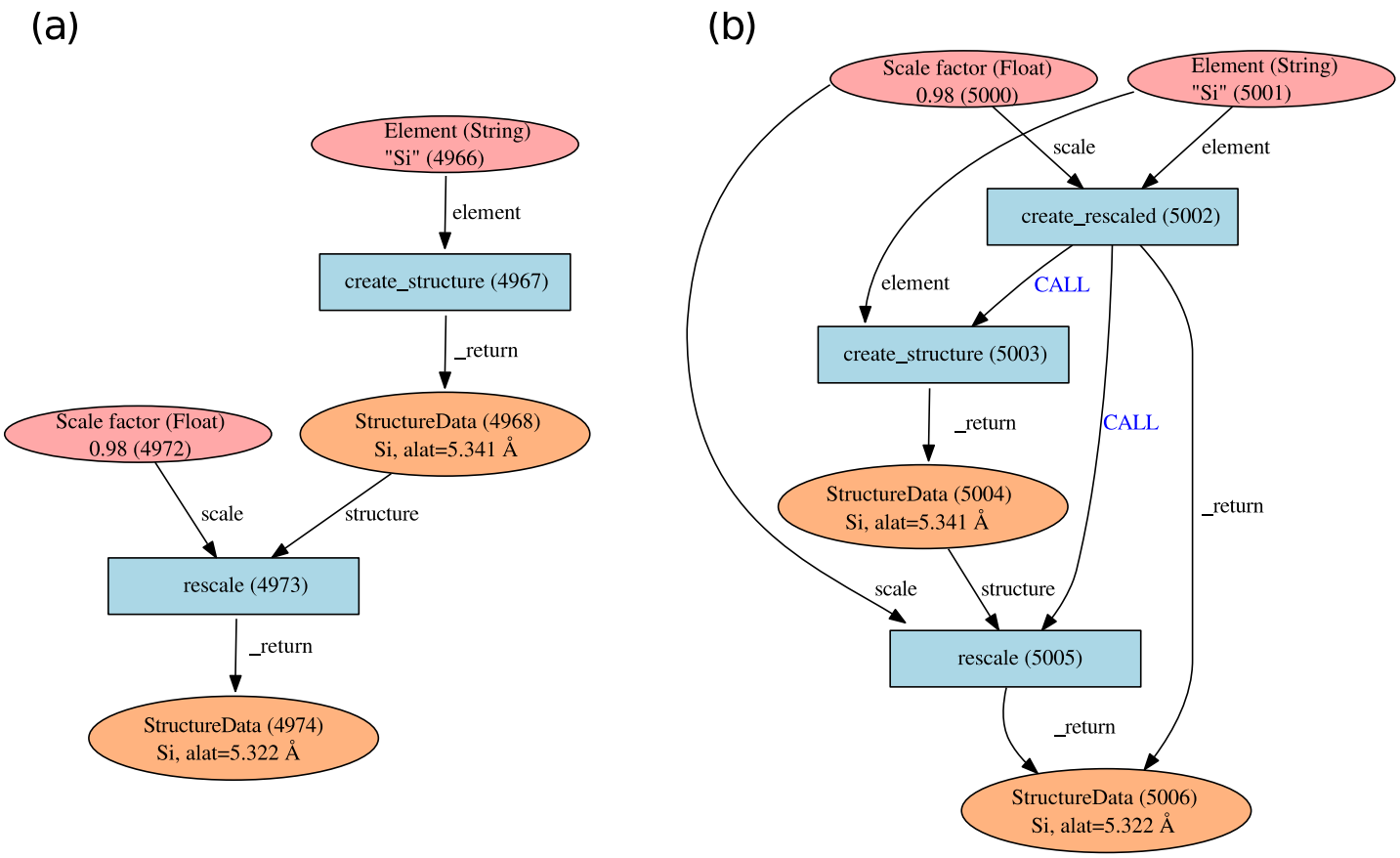

Figure 3 – Typical graphs created by using a workfunction. (a) The workfunction "create_structure" takes a Str object as input and returns a single StructureData object which is used as input for the workfunction "rescale" together with a Float object. This latter workfunction returns another StructureData object, defining a crystal having the rescaled lattice constant. (b) Graph generated by nesting workfunctions. A wrapper workfunction "create_rescaled" calls serially "create_structure" and "rescale". This relationship is stored via "CALL" links.

```
    :return: The structure.
    """
    import numpy as np
    elem_alat= {
                "Si": 5.431, # Angstrom
                "Ge": 5.658,
               }

    # Validate input element
    symbol = str(element)
    if symbol not in elem_alat.keys():
       raise ValueError("Valid elements are only Si and Ge")

    # Create cel starting having lattice parameter alat corresponding to the element
    alat = elem_alat[symbol]
    the_cell = np.array([[0., 0.5, 0.5],
                         [0.5, 0., 0.5],
                         [0.5, 0.5, 0.]]) * alat

    # Create a structure data object
    StructureData = DataFactory("structure")
    structure = StructureData(cell=the_cell)
    structure.append_atom(position=(0., 0., 0.), symbols=str(element))
    structure.append_atom(position=(0.25*alat, 0.25*alat, 0.25*alat),
```

```
                            symbols=str(element))
    return structure
```

For the equation of state you need another function that takes as input a StructureData object and a rescaling factor, and returns a StructureData object with the rescaled lattice parameter (you will find this function in the same file `create_rescale.py` on your virtual machine):

```
def rescale(structure, scale):
    """
    Workfunction to rescale a structure

    :param structure: An AiiDA structure to rescale
    :param scale: The scale factor (for the lattice constant)
    :return: The rescaled structure
    """
    the_ase = structure.get_ase()
    new_ase = the_ase.copy()
    new_ase.set_cell(the_ase.get_cell() * float(scale), scale_atoms=True)
    new_structure = DataFactory('structure')(ase=new_ase)
    return new_structure
```

In order to generate the rescaled starting structures, say for five different lattice parameters you would combine the two functions. Enter the following commands in the `verdi shell` from the `tutorial_scripts` folder.

```
from create_rescale import create_diamond_fcc, rescale

s0 = create_diamond_fcc("Si")
rescaled_structures = [rescale(s0, factor) for factor
                       in (0.98, 0.99, 1.0, 1.1, 1.2)]
```

and store them in the database:

```
s0.store()
for struct in rescaled_structures:
   struct.store()
```

Run the commands above to store all the structures.

As expected, all the structures that you have created are not linked in any manner as you can verify via the `get_inputs()/get_outputs()` methods of the StuctureData class. Instead, you would like these objects to be connected as sketched in Fig. 3a. Now that you are familiar with AiiDA, you know that the way to connect two data nodes is through a calculation. In order to "wrap" python functions and automate the generation of the needed links, in AiiDA we provide you with what we call "workfunctions". A normal function can be converted to a workfunction by using the `@workfunction` decorator[8] that takes care of storing the execution as a calculation and adding the links between the input and output data nodes.

In our case, what you need to do is to modify the two functions as follows (note that we import `workfunction` as `wf` to be shorter, but this is not required). You can do it in the file `create_rescale.py`:

```
# Add this import
from aiida.work.workfunction import workfunction as wf
```

[8] In simple (or even simplified) words, a decorator is a function that modifies the behavior of another function. In python, a function can be decorated by adding a line of the form `@decorating_function_name` on the line just before the `def` line of the decorated function. If you want to know more, there are many online resources explaining python decorators.

```
# Add decorators
@wf
def create_diamond_fcc(element):
    ...
    ...

@wf
def rescale(structure, scale):
    ...
    ...
```

Important: when you use workfunctions, you have to make sure that their input and output are actually Data nodes, so that they can be stored in the database. AiiDA objects such as StructureData, ParameterData, etc. carry around information about their provenance as stored in the database. This is why we must use the special database-storable types Float, Str, etc. as shown in the snippet below.

Try now to run the following script:

```
from aiida.orm.data.base import Float, Str
from create_rescale import create_diamond_fcc, rescale

s0 = create_diamond_fcc(Str("Si"))
rescaled_structures = [rescale(s0,Float(factor)) for factor
                       in (0.98, 0.99, 1.0, 1.1, 1.2)]
```

and check now that the output of `s0` as well as the input of the rescaled structures point to an intermediate ProcessCalculation node, representing the execution of the workfunction, see Fig. 3.

For instance, you can check that the output links of `s0` are the five `rescale` calculations:

```
s0.get_outputs()
```

which outputs

```
[<WorkCalculation: uuid: 01b0b137-974c-4d80-974f-ea4978b12019 (pk: 4970)>,
 <WorkCalculation: uuid: 1af5ead6-0ae0-42a7-969c-1f0e88300f4a (pk: 4973)>,
 <WorkCalculation: uuid: 22dee9d5-0382-48a3-9319-e800506946f1 (pk: 4976)>,
 <WorkCalculation: uuid: dc4c93b7-3e7a-4f51-8d44-cf15c5707ddb (pk: 4979)>,
 <WorkCalculation: uuid: f5b4e9f2-0d50-4b3d-a76b-7a12d232ea54 (pk: 4982)>]
```

and the inputs of each ProcessCalculation ("rescale") are obtained with:

```
for s in s0.get_outputs():
     print s.get_inputs()
```

that will return

```
[0.98, <StructureData: uuid: 9b76b5fa-2908-4f88-a4fb-7a9aa343a1f3 (pk: 4968)>]
[0.99, <StructureData: uuid: 9b76b5fa-2908-4f88-a4fb-7a9aa343a1f3 (pk: 4968)>]
[1.0, <StructureData: uuid: 9b76b5fa-2908-4f88-a4fb-7a9aa343a1f3 (pk: 4968)>]
[1.1, <StructureData: uuid: 9b76b5fa-2908-4f88-a4fb-7a9aa343a1f3 (pk: 4968)>]
[1.2, <StructureData: uuid: 9b76b5fa-2908-4f88-a4fb-7a9aa343a1f3 (pk: 4968)>]
```

5.2.1 Workfunction nesting

One key advantage of workfunctions is that they can be nested, namely, a workfunction can invoke workfunctions inside its definition, and this "call" relationship will also be automatically recorded in the database. As an example, let us combine the two previously defined workfunctions by means of a wrapper workfunction called "create_rescaled" that takes as input the element and the rescale factor.

> Type in your shell (or modify the functions defined in `create_rescale.py` and then run):

```
@wf
def create_rescaled(element, scale):
    """
    Workfunction to create and immediately rescale
    a crystal structure of a given element.
    """
    s0 = create_diamond_fcc(element)
    return rescale(s0,scale)
```

and create an already rescaled structure by typing

```
s1 = create_rescaled(element=Str("Si"), scale=Float(0.98))
```

> Now inspect the input links of `s1`:

```
In [6]: s1.get_inputs()
Out[6]:
[<WorkCalculation: uuid: a672317b-3091-4135-9d84-12c2fff34bfe (pk: 5005)>,
 <WorkCalculation: uuid: a672317b-3091-4135-9d84-12c2fff34bfe (pk: 5005)>,
 <WorkCalculation: uuid: f64f4a70-70ff-4551-ba4d-c186328d8bd6 (pk: 5002)>]
```

The object `s1` has three incoming links, corresponding to *two* different calculations as input (in this case, pks 5002 and 5005). These correspond to the calculations "create_rescaled" and "rescale" as shown in Fig. 3b. It is normal that calculation 5005 has two links, don't worry about that[9]. To see the "call" link, inspect now the outputs of the calculation appearing only once in the list. Write down its <pk> (in general, it will be different from 5002), then in the shell load the corresponding node and inspect the outputs:

```
In [12]: p1 = load_node(<pk>)
In [13]: p1.get_outputs_dict()
```

> You should be able to identify the two "children" calculations as well as the final structure (you will see the calculations linked via CALL links: these are calculation-to-calculation links representing the fact that `create_rescaled` called two sub-workfunctions). The graphical representation of what you have in the database should match Fig. 3b.

5.3 Run a simple workflow

Let us now use the workfunctions that we have just created to build a simple workflow to calculate the equation of state of silicon. We will consider five different values of the lattice parameter obtained rescaling the experimental minimum, $a = 5.431$ Å, by a factor in $[0.98, 0.99, 1.0, 1.02, 1.04]$. We will write a simple script that runs a series of

[9] If you are curious: the two links have the same label, but are of different *link_type*: one is a **create** link, that keeps track of the calculation that actually generated the node. Instead the other one is of type **return**, stating that the workfunction, beside creating that node, also returned it as an output. Calculation 5002 instead only returned the node but it did not generate it, therefore there is only one link between it and the final StructureData.

five calculations and at the end returns the volume and the total energy corresponding to each value of the lattice parameter. For your convenience, besides the functions that you have written so far in the file `create_rescale.py`, we provide you with some other utilities to get the correct pseudopotential and to generate a pw input file, in the module `common_wf.py` which has been put in the `tutorial_scripts` folder.

> We have already created the following script named `simple_sync_workflow.py`, which you are free to look at but please go through the lines carefully and make sure you understand them. If you decide to create your own new script, make sure to also place it in the folder `tutorial_scripts`, otherwise the imports won't work.

Besides the functions in the local folder

```
from create_rescale import create_diamond_fcc, rescale
from common_wf import get_pseudos, generate_scf_input_params
```

you need to import few further AiiDA classes and functions:

```
from aiida.work.run import run
from aiida.work.workfunction import workfunction as wf
from aiida.orm.data.base import Str, Float
from aiida.orm.calculation.job.quantumespresso.pw import PwCalculation
from aiida.work.process_registry import ProcessRegistry
from aiida.orm import DataFactory
```

The only imported function that deserves an explanation is `run`. For the time being, you just need to know that it is a function that needs to be used to execute a new workflow. The actual body of the script is the following. We suggest that you first have a careful look at it before running it.

```
scale_facs = (0.96, 0.98, 1.0, 1.02, 1.04)
labels = ["c1", "c2", "c3", "c4", "c5"]

@wf
def run_eos_wf(codename, pseudo_family, element):
    print "Workfunction node identifiers: {}".format(ProcessRegistry().current_calc_node)
    #Instantiate a JobCalc process and create basic structure
    JobCalc = PwCalculation.process()
    s0 = create_diamond_fcc(Str(element))

    calcs = {}
    for label, factor in zip(labels, scale_facs):
        s = rescale(s0,Float(factor))
        inputs = generate_scf_input_params(s, str(codename), str(pseudo_family))
        print "Running a scf for {} with scale factor {}".format(element, factor)
        result = run(JobCalc,**inputs)
        calcs[label] = get_info(result)

    eos = []
    for label in labels:
        eos.append(calcs[label])

    #Return information to plot the EOS
    ParameterData = DataFactory("parameter")
    retdict = {
            'initial_structure': s0,
            'result': ParameterData(dict={'eos_data': eos})
        }

    return retdict
```

where the utility function `get_info` reads:

```
def get_info(calc_results):
    return (calc_results['output_parameters'].dict.volume,
            calc_results['output_parameters'].dict.energy,
            calc_results['output_parameters'].dict.energy_units)
```

If you look into the previous snippets of code, you will notice that the way we submit a QE calculation is slightly different from what you have seen in the first part of the tutorial. This is required to properly embed individual calculations in a workflow. In this respect, the main required changes are the instructions

```
JobCalc = PwCalculation.process()
```

which instantiates a ProcessCalculation derived from a PwCalculation, and

```
result = run(JobCalc, **inputs)
```

which, when fed with a ProcessCalculation object, submits it to the daemon, waits for its completion and returns the output in the user-defined variable `result`. The latter is a dictionary whose values are the output nodes generated by the calculation, with the link labels as keys. For example, once the calculation is finished, in order to access the total energy, we need to access the ParameterData node which is linked via the "output_parameters" link (see again Fig. 1 of Day 1 Tutorial, to see inputs and outputs of a Quantum ESPRESSO calculation). Once the right node is retrieved as `result['output_parameters']`, we need to get the `energy` attribute. The global operation is achieved by the command

```
result['output_parameters'].dict.energy
```

As you see, the function `run_eos_wf` has been decorated as a workfunction to keep track of the provenance. Finally, in order to get the <pk> associated to the workfunction (and print on the screen for our later reference), we have used the following command to get the node corresponding to the ProcessCalculation:

```
from aiida.work.process_registry import ProcessRegistry
print ProcessRegistry().current_calc_node
```

To run the workflow it suffices to call the function `run_eos_wf` in a python script providing the required input parameters. For simplicity, we have included few lines at the end of the script that invoke the function with a static choice of parameters:

```
def run_eos(codename='pw-5.1@localhost', pseudo_family='GBRV_lda', element="Si"):
    return run_eos_wf(Str(codename), Str(pseudo_family), Str(element))

if __name__ == '__main__':
    run_eos()
```

Run the workflow by running the following command from the `tutorial_scripts` directory:

```
python simple_sync_workflow.py
```

and write down the <pk> of the ProcessCalculation printed on screen at execution.

The command above locks the shell until the full workflow has completed (we will see in a moment how to avoid this). While the calculation is running, you can use (in a different shell) the command `verdi work list` to show ongoing and finished workfunctions. You can "grep" for the <pk> you are interested in. Additionally, you can use the command `verdi work tree <pk>` to show the tree of the sub-workfunctions called by the root workfunction with a given <pk>.

Wait for the calculation to finish, then call the function `plot_eos(<pk>)` that we provided in the file `common_wf.py` to plot the equation of state and fit it with a Birch–Murnaghan equation.

5.4 Run asynchronous functions

You should have noticed that the calculations for different lattice parameters are executed serially, although they might perfectly be executed in parallel because their inputs and outputs are not connected in any way. In the language of workflows, these calculations are executed in a synchronous (or blocking) way, whereas we would like to have them running *asynchronously* (i.e., in a non-blocking way, to run them in parallel). One way to achieve this is to use the `async` function. Make a copy of the script `simple_sync_workflow.py` that we worked on in the previous section and name it `simple_async_workflow.py`. To make the new script work asynchronously, simply change the following subset of lines:

```
from aiida.work.run import run
[...]
for label, factor in zip(labels, scale_facs):
    [...]
    result = run(JobCalc,**inputs)
    calcs[label] = get_info(result)
[...]
eos = []
for label in labels:
    eos.append(calcs[label])
```

replacing them with

```
from aiida.work.run import async
[...]
for label, factor in zip(labels, scale_facs):
    [...]
    future = async(JobCalc,**inputs)
    calcs[label] = future
[...]
eos = []
for label in labels:
    eos.append(get_info(calcs[label].result()))
```

The main differences are:

- `run` is replaced by `async`
- The return value of `async` is not a dictionary describing the outputs of the calculation, but a *future*[10]
- Each calculation starts in the background and the futures returned by `async` are added to the `calc` dictionary.
- At the end of the loop, when all calculations have been launched with `async`, another loop has to be run to gather all the results. This is achieved via the (blocking) `.result()` method that is implemented in the *future* object. The output of the latter method is exactly the same as the one directly returned by the `run` function. It is indeed a dictionary containing all the output nodes, as seen in the previous (synchronous) example.

In the next section we will show you another way to achieve this, which has the added bonus that it introduces checkpoints in the workfunction, from which the calculation can be resumed should it be interrupted.

After applying the modifications, run the script. You will see that all calculations start at the same time, without waiting for the previous ones to finish.

If in the meantime you run `verdi work tree <pk>`, all five calculations are already shown as output. Also, if you run `verdi calculation list`, you will see how the calculations are submitted to the scheduler.

[10] A future is a standard concept in programming. In simple words, you can think of it as a placeholder of the calculation, with a special blocking method to wait for the time when the actual result will be available.

5.5 Workchains, or how not to get lost if your computer shuts down or crashes

The simple workflows that we have used so far have been launched by a python script that needs to be running for the whole time of the execution, namely the time in which the calculation are passed to the daemon, the time required for the daemon to submit the calculation, and the actual time needed by Quantum ESPRESSO to accomplish the calculation. If you had killed the main python process during this time, the workflow would have not terminated correctly. Perhaps you did kill the calculation and you experienced the unpleasant consequences: intermediate calculation results are potentially lost and it is extremely difficult to restart a workflow from the exact place where it stopped.

In order to overcome this limitation, we have implemented into AiiDA a way to insert checkpoints, where the main code defining a workflow can be stopped (you can even shut down the machine on which AiiDA is running!). We call these workfunctions with checkpoints "workchains" because, as you will see, they basically amount to splitting a workfunction in a chain of steps. Each step is then run by the daemon, in a way similar to the remote calculations.

The basic rules that allow you to convert your workfunction-based script to a workchain-based one are listed in Table 5, which focus on the code used to perform the calculation of an equation of state. The modifications needed are put side-to-side to allow for a direct comparison. In the following, when referencing a specific part of the code we will refer to the line number appearing in Table 5.

- Instead of using decorated functions you need to define a class, inheriting from a prototype class called `WorkChain` that is provided by AiiDA (line 6)

- Within your class you need to implement a `define` classmethod that always takes `cls` and `spec` as inputs. (lines 7–8). Here you specify the main information on the workchain, in particular:

  - the *inputs* that the workfunction expects. This is obtained by means of the spec.input() method, which provides as key feature the automatic validation of the input types via the `valid_type` argument (lines 10–12). The same holds true for outputs, as you can use the `spec.output()` method to state what output types are expected to be returned by the workfunction. Both `spec.input()` and `spec.output()` methods are optional, and if not specified, the workfunction will accept any set of inputs and will not perform any check on the outputs. However, to tell that the outputs are not pre-defined, you need to call `spec.dynamic_output()` (line 17).
  - the `outline` consisting in a list of "steps" that you want to run, put in the right sequence (lines 13–16). This is obtained by means of the method `spec.outline()` which takes as input the steps. *Note*: in this example we just split the main execution in two sequential steps, that is, first `run_pw` then `return_results`. However, more complex logic is allowed, as will be explained in the Sec. D.

- You need to split your main code into methods, with the names you specified before into the outline (`run_pw` and `return_results` in this example, lines 19 and 37). Where exactly should you split the code? Well, the splitting points should be put where you would normally block the execution of the script for collecting results in a standard workfunction, namely whenever you call the method `.result()`. Each method should accept only one parameter, `self`, e.g. `def step_name(self)`.

- You will notice that the methods reference the attribute `ctx` through `self.ctx`, which is called the *context* and is inherited from the base class `WorkChain`. A python function or workfunction normally just stores variables in the local scope of the function. For instance, in the example of the subsection 5.3, you stored the `calc_results` in the `eos` list, that was a local variable. In workchains, instead, to preserve variables between different steps, you need to store them in a special dictionary called *context*. As explained above, the context variable `ctx` is inherited from the base class `WorkChain`, and at each step method you just need to update its content. AiiDA will take care of saving the context somewhere between workflow steps (on disk, in the database, ..., depending on how AiiDA was configured). For your convenience, you can also access the value of a context variable as `self.ctx.varname` instead of `self.ctx['varname']` (see e.g. lines 21, 22, 26, 40, 44).

- Any submission within the workflow should not call `run` or `async`, but `submit` to which you have to pass the ProcessCalculation, and a dictionary of inputs (line 30).

Table 5: Side-to-side comparison of the EOS workflow using standard workfunctions (left panel) or "workchains" (right panel).

Workfunctions

```
from aiida.work.run import async
...

@wf
def run_eos_wf(code, pseudo_family, element):
    ...
    s0 = create_diamond_fcc(Str(element))

    calcs = {}
    for label, factor in zip(labels, scale_facs):
        s = rescale(s0,Float(factor))
        inputs = generate_scf_input_params(
            s, str(code), str(pseudo_family))
          ...
        future = async(JobCalc,**inputs)
        calcs[label] = future

    eos=[]
    for label in labels:
        eos.append(get_info(calcs[label].result()))
    #Return information to plot the EOS
    ParameterData = DataFactory("parameter")
    retdict = {
            'initial_structure': s0,
            'result': ParameterData(dict={'eos_data': eos})
     }

    return retdict
```

Workchains

```
from aiida.work.workchain import WorkChain, \
    ToContext, Outputs
from aiida.work.run import submit
...

class EquationOfStates(WorkChain):
    @classmethod
    def define(cls, spec):
        super(EquationOfStates, cls).define(spec)
        spec.input("element", valid_type=Str)
        spec.input("code", valid_type=Str)
        spec.input("pseudo_family", valid_type=Str)
        spec.outline(
            cls.run_pw,
            cls.return_results,
        )
        spec.dynamic_output()

    def run_pw(self):
        ...
        self.ctx.s0 = create_diamond_fcc(Str(self.inputs.element))
        self.ctx.eos_names = []

        calcs = {}
        for label, factor in zip(labels, scale_facs):
            s = rescale(self.ctx.s0,Float(factor))
            inputs = generate_scf_input_params(
              s, str(self.inputs.code), str(self.inputs.pseudo_family))
            ...
            future = submit(JobCalc, **inputs)
            calcs[label] = Outputs(future)

        # Ask the workflow to continue when the results are ready
        # and store them in the context
        return ToContext(**calcs)

    def return_results(self):
        eos = []
        for label in labels:
            eos.append(get_info(self.ctx[label]))
        #Return information to plot the EOS
        ParameterData = DataFactory("parameter")
        retdict = {
                'initial_structure': self.ctx.s0,
                'result': ParameterData(dict={'eos_data': eos})
           }
        for link_name, node in retdict.iteritems():
            self.out(link_name, node)
```

- We need to "wrap" the returned `future` in an `Outputs` object (see line 31). This is necessary because in order to return the futures to the context, one needs to return a `ToContext` object which takes a key-value list of `Outputs` objects (see line 35). By doing this, the workfunction will implicitly wait for the results of all the futures you have specified, and then call the next step *only when all futures have completed*. Importantly, at the beginning of the next step, the results of the futures are stored automatically in the context at the labels that we specified.

- *Return values*: While in a normal workfunction you attach output nodes to the WorkCalculation by invoking the *return* statement, in a workchain you need to call `self.out(link_name, node)` for each node you want to return (line 47-48). Of course, if you have already prepared a dictionary of outputs, you can just use the following syntax:

  ```
  for link_name, node in retdict.iteritems():
       self.out(link_name, node)
  ```

  The advantage of this different syntax is that you can start emitting output nodes already in the middle of the execution, and not necessarily at the very end as it happens for normal functions (*return* is always the last instruction executed in a function). Also, note that once you have called `.out(link_name, node)` on a given `link_name`, you can no longer call `.out()` on the same `link_name`: this will raise an exception.

Inspect the example in the table that compares the two versions of workfunctions to understand in detail the different syntaxes.

Finally, the workflow has to be run. For this you have to use the function `run` passing as arguments the EquationOfStates class and the inputs as key-value arguments. For example, you can execute

```
run(EquationOfStates, element=Str(Si"), code=Str(pw-5.1@localhost"),
    pseudo_family=Str(GBRV_lda"))
```

While the workflow is running, you can check (in a different terminal) what is happening to the calculations using `verdi calculation list`. You will see that after a few seconds the calculations are all submitted to the scheduler and can potentially run at the same time.

Note: As a good practice, you should try to keep the steps as short as possible in term of execution time. The reason is that the daemon can be stopped and restarted only between execution steps and not if a step is in the middle of a long execution.

Finally, as an optional exercise if you have time, you can jump to the Appendix D, which shows how to introduce more complex logic into your WorkChains (if conditionals, while loops etc.). The exercise will show how to realize a convergence loop to obtain the minimum-volume structure in a EOS using the Newtow's algorithm.

Appendices

The following appendices consist of optional exercises, and are mentioned in earlier parts of the tutorial. Go through them only if you have time.

A Calculation input validation

This appendix shows additional ways to debug possible errors with QE, how to use a useful tool that we included in AiiDA to validate the input to Quantum ESPRESSO (and possibly suggest the correct name to mispelled keywords)

There are various reasons why you might end up providing a wrong input to a Quantum-ESPRESSO calculation. Let's check for example this input dictionary, where we inserted two mistakes:

```
parameters_dict = {
    "CTRL": {
        "calculation": "scf",
        "restart_mode": "from_scratch",
```

```
    },
    "SYSTEM": {
        "nat": 2,
        "ecutwfc": 30.,
        "ecutrho": 200.,
    },
    "ELECTRONS": {
        "conv_thr": 1.e-6,
    }
}
```

The two mistakes in the dictionary are the following. First, we wrote a wrong namelist name ('CTRL' instead of 'CONTROL'). Second, we inserted the number of atoms explicitly: while that is how the number of atoms is specified in Quantum ESPRESSO, in AiiDA this key is reserved by the system: in fact this information is already contained in the StructureData. Modify the script with this ParameterData. In this case, we use a tool (the input validator that we provide in AiiDA) to check the input file before submitting. Therefore, the behavior will be slightly different from the previous example: the plugin will check for some keys and will refuse to submit the calculation. This kind of mistakes is revealed by either

- Submitting a calculation. You will see the calculation ending up in the SUBMISSIONFAILED status. Check therefore the logs to recognize the source of the error.
- Submitting a test. You will not be able to successfully create the test and the traceback will guide you to the problem.

Over the time this kind of trivial mistakes can be annoying, but they can be avoided with a utility function that checks the "grammar" of the input parameters. In your script, after you defined the parameters_dict, you can validate it with the command:

```
validated_dict = calc.input_helper(parameters_dict, structure=s)
parameters = ParameterData(dict=validated_dict)
```

The `input_helper` method will check for the correctness of the input parameters. If misspelling or incorrect keys are detected, the method raises an exception, which stops the script before submitting the calculation and thus allows for a more effective debugging.

With this utility, you can also provide a list of keys, without providing the namelists (useful if you don't remember where to put some variables in the input file). Hence, you can provide to `input_helper()` a dictionary like this one:

```
parameters_dict = {'calculation': 'scf',
                   'tstress': True,
                   'tprnfor': True,
                   'ecutwfc': 30.,
                   'ecutrho': 200.,
                   'conv_thr': 1.e-6,
                  }
validated_dict = calc.input_helper(
    parameters_dict, structure=s, flat_mode=True)
```

The validated dictionary will look like:

```
validated_dict
{"CONTROL": {"calculation": "scf",
             "tstress": True,
             "tprnfor": True,
             },
 "SYSTEM": {"ecutwfc": 30.,
            "ecutrho": 200.,
```

```
            },
 "ELECTRONS": {"conv_thr": 1.e-6,
               }
 }
```

B Restarting calculations

Up to now, we have only presented cases in which we were passing wrong input parameters to the calculations, which required us to modify the input scripts and relaunch calculations from scratch. There are several other scenarios in which, more generally, we need to restart calculations from the last step that they have executed. For example when we run molecular dynamics, we might want to add more time steps than we initially thought, or as another example you might want to refine the relaxation of a structure with tighter parameters.

In this section, you will learn how to restart and/or modify a calculation that has run previously. As an example, let us first submit a total energy calculation using a parameters dictionary of the form:

```
parameters_dict = {'CONTROL': {'calculation': 'scf',
                               'tstress': True,
                               'tprnfor': True,
                               },
                   'SYSTEM': {'ecutwfc': 30.,
                              'ecutrho': 200.,
                              },
                   'ELECTRONS': {'conv_thr': 1.e-14,
                                 'electron_maxstep': 3,
                                 },
                    }
```

and submit the calculation with this input. In this case, we set a very low number of self consistent iterations (3), too small to be able to reach the desired accuracy of 10^{-14}: therefore the calculation will not reach a complete end and will be flagged in a FAILED state. However, there is no mistake in the parameter dictionary.

Now, create a new script file, where you will try to restart and correct the input dictionary. We first load the calculation that has just failed (let's call it `c1`)

```
c1 = load_node(PK)
```

(take care of using the correct PK). Then, create a new calculation `c2` which is the copy of the previous one.

```
c2 = c1.create_restart(use_output_structure=False,
                       restart_from_beginning=True,
                       force_restart=True)
```

The flag usage is:

- `use_output_structure`: if True and `c1` has an output structure, `c2` will use it;
- `restart_from_beginning`: if False `c2` will start from the charge density of `c1`, starts from the beginning otherwise;
- `force_restart`: if True `c2` will be created even if `c1` is not in a FINISHED state.

Since this calculation has exactly the same parameters of before, we have to modify the input parameters and increase `electron_maxstep` to a larger value. To this aim, let's load the dictionary of values and change it

```
old_parameters = c2.inp.parameters
parameters_dict = old_parameters.get_dict()
parameters_dict['ELECTRONS']['electron_maxstep'] = 100
```

Note that you cannot modify the `old_parameters` object: it has been used by calculation `c1` and is saved in the database; hence a modification would break the provenance. We have to create a new ParameterData and pass it to c2:

```
new_parameters = ParameterData(dict=parameters_dict)
c2.use_parameters(new_parameters)
```

Now you can launch the new calculation

```
c2.store_all()
c2.submit()
```

that this time can proceed until the end and return converged total energy. Using the restart method, the script is much shorter than the one needed to launch a new one from scratch: you didn't need to define pseudopotentials, structures and k-points, which are the same as before.

Alternatively, you can change the parameters to `create_restart` to run an actual restart with Quantum ESPRESSO.

C Queries in AiiDA - Optional examples and exercises

Optional exercises on relationships (Task 3)

Hint for the exercises:

- You can have projections on properties of more than one entity in your query. You just have to add the *project* key (specifying the list of properties that you want to project) along with the corresponding entity when you append it.

Exercises:

Try to write the following queries:

- Find all descendants of a StructureData with a certain uuid. Print both the StructureData and the descendant.
- Find all the FolderData created by a specific user.

Optional exercises on attributes and extras (Task 4)

Hint for the exercises:

- You can easily order or limit the number of the results by using the *order_by()* and *limit()* methods of the QueryBuilder. For example, we can order all our job calculation by their *id* and limit the result to the first 10 as follows:

```
qb = QueryBuilder()
qb.append(JobCalculation, tag='calc')
qb.order_by({'calc':'id'})
qb.limit(10)
qb.all()
```

Exercises:

- Write a code snippet that informs you how many pseudopotentials you have for each element.
- Smearing contribution to the total energy for calculations:
  1. Write a query that returns the smearing contribution to the energy stored in some instances of ParameterData.
  2. Extend the previous query to also get the input structures.
  3. Which structures have a smearing contribution to the energy smaller or equal to -0.02?

Summarizing what we learned by now - An example

At this point you should be able to do queries with projections, joins and filters. Moreover, you saw how to apply filters and projections even on attributes and extras. Let's discover the full power of the QueryBuilder with a complex graph query that allows you to project various properties from different nodes and apply different filters and joins.

Imagine that you would like to get the smearing energy for all the calculations that have finished and have a Sn_2O_3 as input. Moreover, besides from the smearing energy, you would like to print the units of this energy and the formula of the structure that was given to the calculation. The graphical representation of this query can be seen in Figure 4 and the actual query follows:

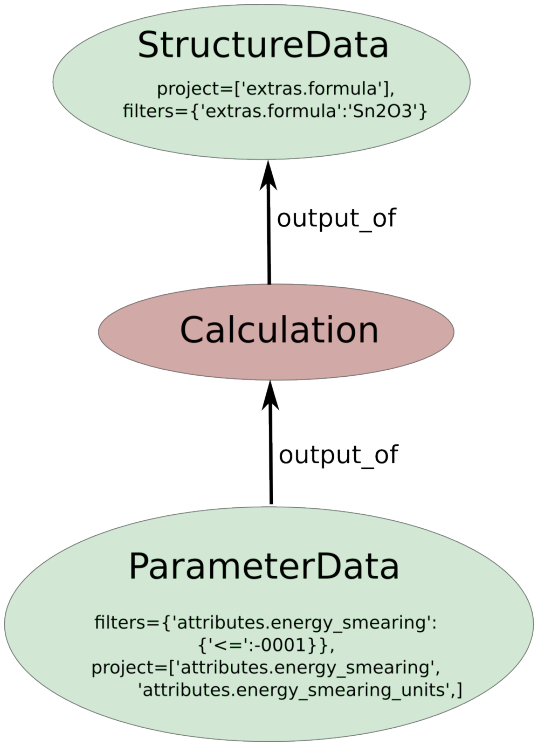

Figure 4 – Complex graph query.

```
qb = QueryBuilder()
qb.append(
        StructureData,
        project=["extras.formula"],
        filters={"extras.formula":"Sn2O3"},
        tag="structure"
    )
qb.append(
        Calculation,
        tag="calculation",
        output_of="structure"
    )
qb.append(
        ParameterData,
        tag="results",
        filters={"attributes.energy_smearing":{"<=":-0.0001}},
        project=[
            "attributes.energy_smearing",
            "attributes.energy_smearing_units",
        ],
```

```
        output_of="calculation"
)
qb.all()
```

D More complex logic in workflows: while loops and conditional statements

In the previous sections, you have been introduced to WorkChains, and the reason for using them over "standard" workfunctions (i.e., functions decorated with `@wf`).

However, in the example of Sec. 5.5, the `spec.outline` was quite simple, with a "static" sequence of two steps. Most often, however, you need dynamic workflows, where you need to decide at runtime whether to continue to compute or not (e.g. in a convergence loop, where you need to stop if convergence has been achieved). To support this scenario, the `spec.outline` can support logic: *while* loops and *if/elif/else* blocks. The simplest way to explain it is to show an example:

```
from aiida.work.workchain import if_, while_

spec.outline(
    cls.s1,
    if_(cls.isA)(
        cls.s2
    ).elif_(cls.isB)(
        cls.s3
    ).else_(
        cls.s4
    ),
    cls.s5,
    while_(cls.condition)(
        cls.s6
    ),
)
```

that would *roughly* correspond, in a python syntax, to:

```
s1()
if isA():
    s2()
elif isB():
    s3()
else:
    s4()
s5()
while condition():
    s6()
```

The only additional thing to know is that conditions (in the example above `isA`, `isB` and `condition`) must be class methods that read the Context `ctx` and return either `True` or `False` depending on whether the condition is met or not.

A suggestion on how to write new workchains: Use the outline to help you in designing the logic. First create the spec.outline writing, almost if you were explaining it in words, what you expect the workflow to do. Then, define one by one the methods. For example, we have prepared a simple workfunction to optimize the lattice parameter of silicon efficiently using a Newton's algorithm on the energy derivative, i.e. the pressure $p = -dE/dV$. You can find it the code at
`tutorial_scripts/pressure_convergence.py`. The outline looks like this:

```
spec.outline(
    cls.init,
    cls.put_step0_in_ctx,
```

```
    cls.move_next_step,
    while_(cls.not_converged)(
        cls.move_next_step,
     ),
    cls.report
)
```

This outline already roughly explains the algorithm: after an initialization (`init`) and putting the first step (number zero) in the ctx (`put_step0_in_ctx`), a function to move to the next step is called (`move_next_step`). This is iterated while a given convergence criterion is not met (`not_converged`). Finally, some reporting is done, including returning some output nodes (`report`).

If you are interested in the details of the algorithm, you can inspect the file. The main ideas are described here:

init Generate a pw calculation for the input structure (with volume V), and one for a structure where the volume is $V + 4\text{Å}^3$ (just to get a closeby volume). Store the results in the context as `r0` and `r1`

put_step0_in_ctx Store in the context V, $E(V)$ and dE/dV for the first calculation `r0`

move_next_step This is the most important function. Calculate V, $E(V)$ and dE/dV for `r1`. Also, estimate d^2E/dV^2 from the finite difference of the first derivative of `r0` and `r1` (helper functions to achieve this are provided). Get the a, b and c coefficients of a parabolic fit $E = aV^2 + bV + c$ and estimated the expected minimum of the EOS function as the minimum of the fit $V_0 = -b/2a$. Finally, replace `r0` with `r1` in the context (i.e., get rid of the oldest point) and launch a new pw calculation at volume V_0, that will be stored in the context replacing `r1`. In this way, at the next iteration `r0` and `r1` will contain the latest two simulations. Finally, at each step some relevant information (coefficients a, b and c, volumes, energies, energy derivatives, ...) are stored in a list called `steps`. This whole list is stored in the context because it provides quantities to be preserved between different workfunction steps.

not_converged Return `True` if convergence has not been achieved yet. Convergence is achieved if the difference in volume between the two latest simulations is smaller than a given threshold (`volume_tolerance`).

report This is the final step. Mainly, we return the output nodes: `steps` with the list of results at each step, and `structure` with the final converged structure.

The results returned in `steps` can be used to represent the evolution of the minimisation algorithm. A possible way to visualize it is presented in Fig. 5, obtained with an initial lattice constant of $a_{\text{lat}} = 5.2\text{Å}$.

References

[1] P. Giannozzi et al., J. Phys. Condens. Matter, 21, 395502 (2009).

[2] S. R. Bahn and K. W. Jacobsen, Comput. Sci. Eng., 4, 56-66 (2002).

[3] S. Ping Ong et al., Comput. Mater. Sci. 68, 314-319 (2013).

[4] K.F. Garrity, J.W. Bennett, K.M. Rabe and D. Vanderbilt, Comput. Mater. Sci. 81, 446 (2014).

[5] I. E. Castelli et al, Standard solid state pseudopotentials (SSSP), in preparation, "http://www.materialscloud.ch/sssp/".

[6] Crystallographic Open Database (COD), "http://www.crystallography.net/cod/".

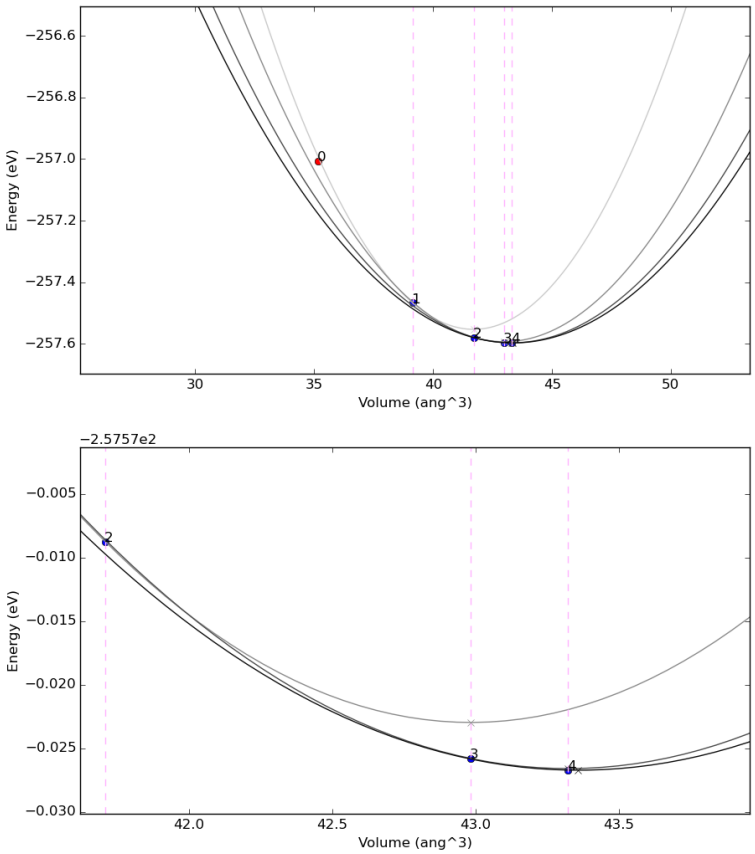

Figure 5 – Example of results of the convergence algorithm presented in Sec. D. The bottom plot is a zoom near the minimum. The dots represent the (volume,energy) points obtained from Quantum ESPRESSO, and the numbers indicate at which iteration they were obtained. The parabolas represent the parabolic fits used in the algorithm; the minimum of the parabola is represented with a small cross, in correspondence of the vertical lines, used as the volume for the following step.