aiida-crystal17

Release 0.4.0

Apr 23, 2019

Contents

`aiida-crystal17` is available at http://github.com/chrisjsewell/aiida-crystal17

CHAPTER 1

User guide

1.1 Getting started

We show below a number of tutorials for the main CRYSTAL17 code that will guide you through submitting your calculations using AiiDA.

Note: these tutorials assume you already installed AiiDA and properly configured AiiDA and configured one CRYSTAL17 code. You can check the main AiiDA-core documentation for more information on how to perform these steps.

1.1.1 Installation

To install from conda (recommended):

```
>> conda install -c conda-forge aiida-crystal17
>> conda install -c bioconda chainmap==1.0.2
```

To install from pypi:

```
>> pip install aiida-crystal17
```

To install the development version:

```
>> git clone https://github.com/chrisjsewell/aiida-crystal17 .
>> cd aiida-crystal17
>> pip install -e .  # also installs aiida, if missing (but not postgres)
#>> pip install -e .[pre-commit,testing] # install extras for more features
>> verdi quicksetup  # better to set up a new profile
>> verdi calculation plugins  # should now show your calclulation plugins
```

Then use `verdi code setup` with a `crystal17.` input plugin to set up an AiiDA code for that plugin.

1.2 Basic Calculation Plugin

The `crystal17.basic` plugin is the simplest calculation plugin. It takes a pre-written .d12 file as input and (optionally) a .gui file with geometry, for .d12 inputs containing the `EXTERNAL` keyword.

1.2.1 Command Line Example

An example script is available within the examples folder. You also either need to have the `runcry17` executable available locally or set the global variable `export MOCK_EXECUTABLES=true` to use a dummy executable. Then, assuming AiiDA is configured and your database is running, the script can be run within a terminal:

```
>> verdi daemon start         # make sure the daemon is running
>> cd examples
>> verdi run test_submit_basic.py       # submit test calculation
submitted calculation; calc=Calculation(PK=5)
>> verdi calculation list -a  # check status of calculation
  PK  Creation    State           Sched. state    Computer    Type
----  ----------  --------------  -------------  ----------  -------------------------
↪--
5     1m ago      WITHSCHEDULER                  localhost   crystal17.basic
>> verdi calculation list -a  # after a few seconds
  PK  Creation    State           Sched. state    Computer    Type
----  ----------  --------------  -------------  ----------  -------------------------
↪--
5     1m ago      FINISHED        DONE           localhost   crystal17.basic
```

Once the calculation has run, it will be linked to the input nodes and a number of output nodes:

```
>> verdi calculation show 5
-----------  ------------------------------------------------
type         CryBasicCalculation
pk           5
uuid         3d9f804b-84db-443a-b6f8-69c15d96d244
label        aiida_crystal17 test
description  Test job submission with the aiida_crystal17 plugin
ctime        2018-08-27 15:23:38.670705+00:00
mtime        2018-08-27 15:24:26.516127+00:00
computer     [2] localhost
code         runcry17
-----------  ------------------------------------------------
##### INPUTS:
Link label       PK    Type
---------------  ----  --------------
input_external   3     SinglefileData
input_file       4     SinglefileData
##### OUTPUTS:
Link label           PK  Type
-----------------  ----  -------------
remote_folder       6    RemoteData
retrieved           7    FolderData
output_parameters   8    ParameterData
output_settings     9    StructSettingsData
output_structure    10   StructureData
##### LOGS:
There are 1 log messages for this calculation
Run 'verdi calculation logshow 5' to see them
```

The outputs represent:

- `remote_folder` provides a symbolic link to the work directory where the computation was run.
- `retrieved` stores a folder containing the full main output of `runcry17` (as main.out)
- `output_parameters` stores a dictionary of key parameters in the database, for later querying.
- `output_structure` stores the final geometry from the calculation
- **`output_settings` stores additional information on the structure,** such as the symmetry operations.

For compatibility, parameters are named with the same convention as `aiida-quantumespresso`:

```
>> verdi data parameter show 8
{
  "calculation_spin": false,
  "calculation_type": "restricted closed shell",
  "ejplugins_version": "0.9.7",
  "energy": -7380.22160519032,
  "energy_units": "eV",
  "errors": [],
  "mulliken_charges": [
    0.776999999999999,
    -0.776999999999999
  ],
  "mulliken_electrons": [
    11.223,
    8.777
  ],
  "number_of_assymetric": 2,
  "number_of_atoms": 2,
  "parser_class": "CryBasicParser",
  "parser_version": "0.3.0a0",
  "parser_warnings": [
    "no initial structure available, creating new kinds for atoms"
  ],
  "scf_iterations": 7,
  "volume": 18.65461525,
  "wall_time_seconds": 5,
  "warnings": []
}
```

You can view the structure settings content by (use `-c` to view the symmetry operations):

```
>> verdi data cry17-settings show 9
centring_code: 1
crystal_type:  1
num_symops:    48
space_group:   1
```

The final structure can be directly viewed by a number of different programs (assuming the executables are available):

```
>> verdi data structure show --format xcrysden 10
```

1.3 Main Calculation Plugin

The `crystal17.main` plugin is designed with a more programmatic input interface. It creates the input `.d12` and `.gui` files, from a set of AiiDa `Data` nodes.

Note: The approach mirrors closely that of the `aiida-quantumespresso.pw` plugin, which is discussed in this tutorial

---

Note: See *Main Calculation Immigration* for a method to immigrate existing output/input files as a `crystal17.main` calculation.

This chapter will show how to launch a single CRYSTAL17 calculation. We will look at how to run a computation *via* the terminal, then how to construct the inputs for a computation in Python. It is assumed that you have already performed the installation, and that you already set up a computer (with verdi),installed CRYSTAL17 and the `runcry17` executable on the cluster and in AiiDA. Although the code should be quite readable, a basic knowledge of Python and object programming is useful.

1.3.1 Command Line Interface

Example Script Execution

An example script is available within the examples folder. You also either need to have the `runcry17` executable available locally or set the global variable `export MOCK_EXECUTABLES=true` (to use a dummy executable). Then, assuming AiiDA is configured and your database is running, the script can be run within a terminal:

```
>> verdi daemon start         # make sure the daemon is running
>> cd examples
>> verdi run test_submit_main.py       # submit test calculation
submitted calculation; calc=Calculation(PK=1)
>> verdi calculation list -a  # check status of calculation
  PK  Creation    State           Sched. state    Computer       Type
----  ----------  --------------  -------------  --------------  ------------------
1     1m ago      WITHSCHEDULER   RUNNING        localhost-test  crystal17.main
>> verdi calculation list -a  # after completion (~30 minutes if using runcry17)
  PK  Creation    State           Sched. state    Computer       Type
----  ----------  --------------  -------------  --------------  ------------------
1     4m ago      FINISHED        DONE           localhost-test  crystal17.main
```

Once the calculation has run, it will be linked to the input nodes and a number of output nodes:

```
verdi calculation show 1
-----------  ------------------------------------------------------
type         CryMainCalculation
pk           1
uuid         3d9f804b-84db-443a-b6f8-69c15d96d244
label        aiida_crystal17 test
description  Test job submission with the aiida_crystal17 plugin
ctime        2018-08-27 15:23:38.670705+00:00
mtime        2018-08-27 15:24:26.516127+00:00
computer     [1] localhost-test
code         runcry17
```

(continues on next page)

(continued from previous page)

```
-----------  --------------------------------------------------
##### INPUTS:
Link label       PK  Type
------------  ----  -------------
parameters     4     ParameterData
settings       5     StructSettingsData
basis_Ni       2     BasisSetData
basis_O        3     BasisSetData
structure      6     StructureData
##### OUTPUTS:
Link label            PK  Type
-----------------  ----  -------------
remote_folder       7     RemoteData
retrieved           8     FolderData
output_parameters   9     ParameterData
output_structure    10    StructureData
##### LOGS:
There are 1 log messages for this calculation
Run 'verdi calculation logshow 1' to see them
```

The inputs represent:

- **`parameters` is a dictionary of (structure independent) data,** used to create the main.d12 file.
- `structure` stores the initial atomic configuration for the calculation.
- **`settings` stores additional data related to the initial atomic** configuration, such as symmetry operations and initial spin.
- `basis_` store the basis set for each element

The outputs represent:

- `remote_folder` provides a symbolic link to the work directory where the computation was run.
- `retrieved` stores a folder containing the full stdout of `runcry17` (as main.out)
- `output_parameters` stores a dictionary of key parameters in the database, for later querying.
- `output_structure` stores the final geometry from the calculation

Input and Output Parameters

Both can be viewed at the command line:

```
>> verdi data parameter show 4
{
"geometry": {
  "optimise": {
    "type": "FULLOPTG"
  }
},
"scf": {
  "k_points": [
    8,
    8
  ],
  "numerical": {
    "FMIXING": 30
```

(continues on next page)

(continued from previous page)

```
  },
  "post_scf": [
    "PPAN"
  ],
  "single": "UHF",
  "spinlock": {
    "SPINLOCK": [
      0,
      15
    ]
  }
},
"title": "NiO Bulk with AFM spin"
}
```

For compatibility, output parameters are named with the same convention as in aiida-quantumespresso.pw

```
>> verdi data parameter show 9
{
  "calculation_spin": true,
  "calculation_type": "unrestricted open shell",
  "ejplugins_version": "0.9.7",
  "energy": -85124.8936673389,
  "energy_units": "eV",
  "errors": [],
  "mulliken_spin_total": 0.0,
  "mulliken_spins": [
    3.057,
    -3.057,
    -0.072,
    0.072
  ],
  "number_of_assymetric": 4,
  "number_of_atoms": 4,
  "number_of_symmops": 16,
  "parser_class": "CryBasicParser",
  "parser_version": "0.2.0a0",
  "parser_warnings": [],
  "scf_iterations": 13,
  "volume": 36.099581472,
  "wall_time_seconds": 187,
  "warnings": []
}
```

Input and Output Structures

The structures can be directly opened by a number of different programs (assuming the executables are available):

```
>> verdi data structure show --format xcrysden 10
```

Note: The output structure will only be present for optimisations, and not SCF computations, i.e. only when the input structure has changed

Structure Settings Data

This node contains data to create the main.d12, which is specific to the structure:

```
>> verdi data cry17-settings show -symmetries 5
centring_code:       1
computation_class:   Symmetrise3DStructure
computation_version: 0.3.0a0
crystal_type:        4
kinds:
  spin_alpha: [Ni1]
  spin_beta:  [Ni2]
operations:           [[1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
                     0.0], [-1.0, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0, 0.0, -1.0, 1.0,
                     6.6613e-16, 0.0], [0.0, -1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
                     1.0, 2.2204e-16, 4.4409e-16, 0.0], [0.0, 1.0, 0.0, -1.0,
                     0.0, 0.0, 0.0, 0.0, -1.0, 1.0, 2.2204e-16, 6.163e-33],
                     [-1.0, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 1.0, 1.0,
                     6.6613e-16, 0.0], [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                     -1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, -1.0, 0.0, 0.0, 0.0,
                     0.0, 1.0, 1.0, 2.2204e-16, 6.163e-33], [0.0, -1.0, 0.0,
                     1.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.2204e-16, 4.4409e-16,
                     0.0], [1.0, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0, 0.0, -1.0, 0.0,
                     6.6613e-16, 0.0], [-1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                     1.0, 1.0, 0.0, 0.0], [0.0, -1.0, 0.0, -1.0, 0.0, 0.0, 0.0,
                     0.0, -1.0, 2.2204e-16, 2.2204e-16, 0.0], [0.0, 1.0, 0.0,
                     1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 4.4409e-16, 6.163e-33],
                     [-1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, -1.0, 1.0, 0.0,
                     0.0], [1.0, 0.0, 0.0, 0.0, -1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                     6.6613e-16, 0.0], [0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
                     -1.0, 1.0, 4.4409e-16, 6.163e-33], [0.0, -1.0, 0.0, -1.0,
                     0.0, 0.0, 0.0, 0.0, 1.0, 2.2204e-16, 2.2204e-16, 0.0]]
space_group:          123
symmetry_program:     spglib
symmetry_version:     1.9.10
```

In this case the symmetry operations, have been pre computed by the *Symmetrise3DStructure* workflow, which will be discussed in *Python API Walk-through*.

Basis Sets

Basis sets are stored as individual nodes:

```
>> verdi data cry17-basis show -c 2
atomic_number: 28
author:        John Smith
basis_type:    all-electron
class:         sto3g
element:       Ni
filename:      sto3g_Ni.basis
md5:           fd341c4056cffcbd63ab92a94dea80e4
num_shells:    5
year:          1999
28 5
1 0 3  2.  0.
1 1 3  8.  0.
1 1 3  8.  0.
```

(continues on next page)

(continued from previous page)

```
1 1 3  2.  0.
1 3 3  8.  0.
```

They can also (preferably) be grouped into families:

```
>> verdi data cry17-basis listfamilies
Family      Num Basis Sets
--------  ----------------
sto3g                    3
```

Families can be created from a folder of individual basis set files, optionally with a yaml meta-data header (see *Basis Sets*):

```
>> verdi data cry17-basis uploadfamily --help
Usage: verdi data cry17-basis uploadfamily [OPTIONS]

  Upload a family of CRYSTAL Basis Set files.

Options:
  --path PATH             Path to a folder containing the Basis Set files
  --ext TEXT              the file extension to filter by
  --name TEXT             Name of the BasisSet family  [required]
  -D, --description TEXT  A description for the family
  --stop-if-existing      Abort when encountering a previously uploaded Basis
                          Set file
  --dry-run               do not commit to database or modify configuration
                          files
  --help                  Show this message and exit.
```

1.3.2 Python API Walk-through

Within this demonstration we will show how to use the input nodes can be used to create the following CRYSTAL17 input (and associated external geometry):

```
NiO Bulk with AFM spin
EXTERNAL
END
28 5
1 0 3  2.  0.
1 1 3  8.  0.
1 1 3  8.  0.
1 1 3  2.  0.
1 3 3  8.  0.
8 2
1 0 3  2.  0.
1 1 3  6.  0.
99 0
END
UHF
SHRINK
8 8
ATOMSPIN
2
1 1
2 -1
```

(continues on next page)

(continued from previous page)

```
FMIXING
30
SPINLOCK
0 15
PPAN
END
```

In the old way, not only you had to prepare 'manually' this file, but also prepare the scheduler submission script, send everything on the cluster, etc. We are going instead to prepare everything in a more programmatic way.

We decompose this script into:

1. `parameters` containing aspects of the input which are independent of the geometry.
2. `structure` defining the geometry and species of the unit cell
3. `settings` defining additional geometric and species specific data (such as spin)
4. `basis_sets` defining the basis set for each atomic type

Parameters

The `parameter` input data defines the content in the `.d12` input file, that is **independent of the geometry**. It follows the naming convention and structure described in the CRYSTAL17 Manual.

```
params = {'scf': {'k_points': (8, 8),
                  'numerical': {'FMIXING': 30},
                  'post_scf': ['PPAN'],
                  'single': 'UHF',
                  'spinlock': {'SPINLOCK': (0, 15)}},
          'title': 'NiO Bulk with AFM spin'}

from aiida.orm import DataFactory
ParameterData = DataFactory('parameter')

parameters = ParameterData(dict=params)
```

The only mandated key is `k_points` (known as `SHRINK` in CRYSTAL17), and the full range of allowed keys, and their validation, is available in the inputd12.schema.json, which can be used programmatically:

```
from aiida_crystal17.validation import read_schema, validate_with_json
read_schema("inputd12")
validate_with_json(params, "inputd12")
```

The dictionary can also be written in a flattened manner, delimited by '.', and subsequently converted:

```
params = {
      "title": "NiO Bulk with AFM spin",
      "scf.single": "UHF",
      "scf.k_points": (8, 8),
      "scf.spinlock.SPINLOCK": (0, 15),
      "scf.numerical.FMIXING": 30,
      "scf.post_scf": ["PPAN"]
  }

from aiida_crystal17.utils import unflatten_dict
params = unflatten_dict(params)
```

This dictionary is used to create the outline of the .d12 file:

```
>>> from aiida_crystal17.parsers.inputd12_write import write_input
>>> write_input(params, ["<basis sets>"])
NiO Bulk with AFM spin
EXTERNAL
END
<basis sets>
99 0
END
UHF
SHRINK
8 8
FMIXING
30
SPINLOCK
0 15
PPAN
END
```

Here is a relatively exhaustive parameter dictionary, of the keys implemented thus far:

```
params = {
    "title": "a title",
    "geometry": {
        "info_print": ["ATOMSYMM", "SYMMOPS"],
        "info_external": ["STRUCPRT"],
        "optimise": {
            "type": "FULLOPTG",
            "hessian": "HESSIDEN",
            "gradient": "NUMGRATO",
            "info_print": ["PRINTOPT", "PRINTFORCES"],
            "convergence": {
                "TOLDEG": 0.0003,
                "TOLDEX": 0.0012,
                "TOLDEE": 7,
                "MAXCYCLE": 50,
                "FINALRUN": 4
            },
        }
    },
    "basis_set": {
        "CHARGED": False,
    },
    "scf": {
        "dft": {
            "xc": ["LDA", "PZ"],
            # or
            # "xc": "HSE06",
            # or
            # "xc": {"LSRSH-PBE": [0.11, 0.25, 0.00001]},
            "SPIN": True,
            "grid": "XLGRID",
            "grid_weights": "BECKE",
            "numerical": {
                "TOLLDENS": 6,
                "TOLLGRID": 14,
                "LIMBEK": 400
```

(continues on next page)

(continued from previous page)

```
            }
        },
        # or
        # "single": "UHF",
        "k_points": [8, 8],
        "numerical": {
            "BIPOLAR": [18, 14],
            "BIPOSIZE": 4000000,
            "EXCHSIZE": 4000000,
            "EXCHPERM": False,
            "ILASIZE": 6000,
            "INTGPACK": 0,
            "MADELIND": 50,
            "NOBIPCOU": False,
            "NOBIPEXCH": False,
            "NOBIPOLA": False,
            "POLEORDR": 4,
            "TOLINTEG": [6, 6, 6, 6, 12],
            "TOLPSEUD": 6,
            "FMIXING": 0,
            "MAXCYCLE": 50,
            "TOLDEE": 6,
            "LEVSHIFT": [2, 1],
            "SMEAR": 0.1
        },
        "fock_mixing": "DIIS",
        # or
        # "fock_mixing": {"BROYDEN": [0.0001, 50, 2]},
        "spinlock": {
            "SPINLOCK": [1, 10]
        },
        "post_scf": ["GRADCAL", "PPAN"]
    }
}
```

Structure

The `structure` refers to a standard `StructureData` node in AiiDa. We now proceed in setting up the structure.

Note: Here we discuss only the main features of structures in AiiDA, needed to run a CRYSTAL17 calculation.

For more detailed information, have a look to the AiiDa Tutorial or QuantumEspresso Tutorial.

Structures consist of:

- A cell with a basis vectors and whether it is periodic, for each dimension
- `Site` with a cartesian coordinate and reference to a kind
- `Kind` which details the species and composition at one or more sites

The simplest way to create a structure is *via* `ase`:

```
from ase.spacegroup import crystal
```

(continues on next page)

(continued from previous page)

```
atoms = crystal(
  symbols=[28, 8],
  basis=[[0, 0, 0], [0.5, 0.5, 0.5]],
  spacegroup=225,
  cellpar=[4.164, 4.164, 4.164, 90, 90, 90])

from aiida.orm import DataFactory
StructureData = DataFactory('structure')

structure = StructureData(ase=atoms)
```

As default, one kind is created per atomic species (named as the atomic symbol):

```
>>> structure.get_site_kindnames()
['Ni', 'Ni', 'Ni', 'Ni', 'O', 'O', 'O', 'O']
```

However, we may want to specify more than one kind per species (for example to setup anti-ferromagnetic spin). We can achieve this by tagging the atoms:

```
>>> atoms_afm = atoms.copy()
>>> atoms_afm.set_tags([1, 1, 2, 2, 0, 0, 0, 0])
>>> structure_afm = StructureData(ase=atoms_afm)
>>> structure_afm.get_site_kindnames()
['Ni1', 'Ni1', 'Ni2', 'Ni2', 'O', 'O', 'O', 'O']
```

Structure Settings

Since we **always** use the `EXTERNAL` keyword for geometry, any manipulation to the geometry is undertaken before calling CRYSTAL (i.e. we delegate the responsibility for geometry away from CRYSTAL). Also, we may want to add atom specific inputs to the `.d12` (such as spin).

The `settings` parameters are used to define some key aspects of the atomic configurations:

1. Properties by `Kind`
2. Crystallographic data for the geometry
3. The input symmetry operations

Available validation schema for the settings data can be viewed programattically at *`data_schema`*

Or *via* the command line:

```
>>> verdi data cry17-settings schema
$schema:              http://json-schema.org/draft-04/schema#
additionalProperties: False
properties:
  centring_code:
    description: The crystal type, as designated by CRYSTAL17
    maximum:     6
    minimum:     1
    type:        integer
  computation_class:
    description: the class used to compute the settings
    type:        string
  computation_version:
    description: the version of the class used to compute the settings
```

(continues on next page)

(continued from previous page)

```
    type:        string
  crystal_type:
    description: The crystal type, as designated by CRYSTAL17
    maximum:     6
    minimum:     1
    type:        integer
  kinds:
    additionalProperties: False
    description:          settings for input properties of each species kind
    properties:
      fixed:
        description: kinds with are fixed in position for optimisations (set by
                     FRAGMENT)
        items:
          type:        string
          uniqueItems: True
        type:        array
      ghosts:
        description: kinds which will be removed, but their basis set are left
                     (set by GHOSTS)
        items:
          type:        string
          uniqueItems: True
        type:        array
      spin_alpha:
        description: kinds with initial alpha (+1) spin (set by ATOMSPIN)
        items:
          type:        string
          uniqueItems: True
        type:        array
      spin_beta:
        description: kinds with initial beta (-1) spin (set by ATOMSPIN)
        items:
          type:        string
          uniqueItems: True
        type:        array
    type:                object
  operations:
    description: symmetry operations to use (in the fractional basis)
    items:
      description: each item should be a list of
                   [r00,r10,r20,r01,r11,r21,r02,r12,r22,t0,t1,t2]
      items:
        maximum: 1
        minimum: -1
        type:    number
      maxItems:    12
      minItems:    12
      type:        array
    type:        [null, array]
  space_group:
    description: Space group number (international)
    maximum:     230
    minimum:     1
    type:        integer
  symmetry_program:
    description: the program used to generate the symmetry
```

(continues on next page)

(continued from previous page)

```
    type:        string
  symmetry_version:
    description: the version of the program used to generate the symmetry
    type:        string
required:            [space_group, crystal_type, centring_code, operations]
title:               CRYSTAL17 structure symmetry settings
type:                object
```

Properties by Kind

The *kinds* lists can be populated by kind names. For example, for a stucture with kinds: `['Ni1', 'Ni1', 'Ni2', 'Ni2', 'O', 'O', 'O', 'O', 'S']`, if the kinds settings are:

```
{
  'kinds': {
      'fixed': ['O'],
      'ghosts': ['S'],
      'spin_alpha': ['Ni1'],
      'spin_beta': ['Ni2']
  }
}
```

Then the `main.d12` would contain (assuming we do not create a primitive cell);

```
FRAGMENT
8
1 2 3 4 5 6 7 8
```

in the `OPTGEOM` block (specifying atoms free to move),

```
GHOSTS
1
9
```

In the `BASIS SET` block (specifying atoms which are removed, but their basis sets left), and

```
ATOMSPIN
1 1 2 1 3 1 4 1 5 -1 6 -1 7 -1 8 -1
```

In the `HAMILTONIAN` block (specifying initial spin state)

Symmetry

In the `main.gui` file, as well as using the dimensionality (i.e. periodic boundary conditions), basis vectors and atomic positions, provided by the `structure`, we also need to specify the symmetry operators, and the crystal system and primitive-to-crystallographic transform (referred to as the `CENTRING CODE` in `CRYSTAL`).

These are provided by the `crystal17.structsettings`:

```
{
  'space_group': 2,
  'operations': [
      [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
      [-1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0]
   ],
  'crystal_type': 1,
  'centring_code': 1
}
```

Note: The `operations` are given as a flattened version of the rotation matrix, followed by the translation vector, in **fractional** coordinates.

Pre-Processing of the Structure

To compute the symmetry operations, and optionally convert the structure to a standard primitive, the space group and symmetry operators can be computed internally, a pre-processing workflow has been created (currently only for 3D-periodic structures), *`Symmetrise3DStructure`*, which can be run with the helper function *`run_symmetrise_3d_structure()`*.

This uses the spglib library to compute symmetries, but with the added constraint that sites with the same `Kind` must be symmetrically equivalent.

Important: Symmetrical equivalence is based on atomic number **AND** kind.

So, for example, taking our structure with kinds;

```
['Ni', 'Ni', 'Ni', 'Ni', 'O', 'O', 'O', 'O']
```

```
>>> settings_dict = {'primitive': False, 'standardize': False, 'idealize': False,
... 'kinds': {'fixed': [], 'ghosts': [], 'spin_alpha': [], 'spin_beta': []},
... 'angletol': None, 'symprec': 0.01}

>>> from aiida_crystal17.workflows.symmetrise_3d_struct import run_symmetrise_3d_
↪structure
>>> newstruct, settings = run_symmetrise_3d_structure(structure, settings_dict)
>>> settings.num_symops
192
>>> settings.space_group
225
```

Whereas, for the structure with multiple Ni kinds;

```
['Ni1', 'Ni1', 'Ni2', 'Ni2', 'O', 'O', 'O', 'O']
```

```
>>> newstruct, settings = run_symmetrise_3d_structure(structure_afm, settings_dict)
>>> settings.num_symops
32
>>> settings.space_group
123
```

Since CRYSTAL17 expects the geometry in a standardized form, which minimises the translational symmetry components, he structure can be converted to a standardized, and (optionally) primitive cell:

```
>>> settings_dict = {'primitive': True, 'standardize': True, 'idealize': False,
... 'kinds': {'fixed': [], 'ghosts': [], 'spin_alpha': [], 'spin_beta': []},
... 'angletol': None, 'symprec': 0.01}

>>> newstruct, settings = run_symmetrise_3d_structure(structure, settings_dict)
>>> newstruct.get_formula()
'NiO'
>>> settings.data.centring_code
5
```

```
>>> newstruct, settings = run_symmetrise_3d_structure(structure_afm, settings_dict)
>>> newstruct.get_formula()
'Ni2O2'
>>> settings.data.centring_code
1
```

The other option is to `idealize` the structure, which removes distortions of the unit cell's atomic positions, compared to the ideal symmetry.

Basis Sets

Basis sets are stored as separate *BasisSetData* nodes, in a similar fashion to `UpfData` (discussed in this tutorial). They are created individually from a text file, which contains the content of the basis set and (optionally) a yaml style header section, fenced by `---`:

```
---
author: John Smith
year: 1999
class: sto3g
---
12 3
1 0 3  2.  0.
1 1 3  8.  0.
1 1 3  2.  0.
```

```
>>> import os
>>> from aiida_crystal17.tests import TEST_DIR
>>> fpath = os.path.join(TEST_DIR, "input_files", "sto3g", "sto3g_Mg.basis")

>>> from aiida.orm import DataFactory
>>> BasisSetData = DataFactory("crystal17.basisset")
>>> bset, created = BasisSetData.get_or_create(fpath)
>>> bset.metadata
{
  'num_shells': 3,
  'author': 'John Smith',
  'atomic_number': 12,
  'filename': 'sto3g_Mg.basis',
  'element': 'Mg',
  'year': 1999,
  'basis_type': 'all-electron',
  'class': 'sto3g',
  'md5': '0731ecc3339d2b8736e61add113d0c6f'
}
```

The attributes of the basis set are stored in the database, and the md5 hash-sum is used to test equivalence of two basis sets.

A simpler way to create and refer to basis sets, is *via* a family group. All basis sets in a folder can be read and saved to a named family by:

```
>>> from aiida_crystal17.data.basis_set import upload_basisset_family
>>> nfiles, nuploaded = upload_basisset_family(
        os.path.join(TEST_DIR, "input_files", "sto3g"),
        "sto3g", "group of sto3g basis sets",
        extension=".basis")
```

Basis families can be searched (optionally by the elements they contain):

```
>>> from aiida.orm import DataFactory
>>> basis_cls = DataFactory('crystal17.basisset')
>>> basis_cls.get_basis_groups(["Ni", "O"])
[<Group: "sto3g" [type data.basisset.family], of user test@hotmail.com>]
```

The basis sets for a particular structure are then extracted by `crystal17.main`:

```
>>> from ase.spacegroup import crystal

>>> atoms = crystal(
...   symbols=[28, 8],
...   basis=[[0, 0, 0], [0.5, 0.5, 0.5]],
...   spacegroup=225,
...   cellpar=[4.164, 4.164, 4.164, 90, 90, 90])

>>> from aiida.orm import DataFactory
>>> StructureData = DataFactory('structure')

>>> structure = StructureData(ase=atoms)

>>> from aiida_crystal17.data.basis_set import get_basissets_from_structure
>>> get_basissets_from_structure(structure, "sto3g", by_kind=False)
{'Ni': <BasisSetData: uuid: d1529498-1cc4-48cc-9524-42355e7a6f18 (pk: 2320)>,
'O': <BasisSetData: uuid: 67d87176-cb83-4082-be06-8dae80c488c3 (pk: 2321)>}
```

Important: Unlike aiida-quantumespresso.pw, `crystal17.main` uses one basis sets per atomic number only **NOT** per kind. This is because, using multiple basis sets per atomic number is rarely used in CRYSTAL17, and is limited anyway to only two types per atomic number.

Input Preparation and Validation

Before creating and submitting the calculation, *`CryMainCalculation`* provides a helper function, to prepare the parameter and settings data and validate their content.

```
from aiida.orm import DataFactory, CalculationFactory
StructureData = DataFactory('structure')
calc_cls = CalculationFactory('crystal17.main')

atoms = crystal(
    symbols=[28, 8],
```

(continues on next page)

(continued from previous page)

```
    basis=[[0, 0, 0], [0.5, 0.5, 0.5]],
    spacegroup=225,
    cellpar=[4.164, 4.164, 4.164, 90, 90, 90])
atoms.set_tags([1, 1, 2, 2, 0, 0, 0, 0])
instruct = StructureData(ase=atoms)
settings_dict = {"kinds.spin_alpha": ["Ni1"],
                 "kinds.spin_beta": ["Ni2"]}
newstruct, settings = run_symmetrise_3d_structure(instruct, settings_dict)

params = {
    "title": "NiO Bulk with AFM spin",
    "scf.single": "UHF",
    "scf.k_points": (8, 8),
    "scf.spinlock.SPINLOCK": (0, 15),
    "scf.numerical.FMIXING": 30,
    "scf.post_scf": ["PPAN"]
}

pdata = calc_cls.prepare_and_validate(params, newstruct,
                                      settings,
                                      basis_family="sto3g",
                                      flattened=True)
```

Creating and Submitting Calculation

As in the AiiDa tutorial Setup a code and the qe.pw tutorial, to run the computation on a remote computer, you will need to setup `computer` and `code` nodes. Then the code can be submitted using `verdi run` or programmatically:

```
from aiida import load_dbenv
load_dbenv()

from aiida.orm import Code
code = Code.get_from_string('cry17.2@MyHPC')
calc = code.new_calc()

calc.label = "aiida_crystal17 test"
calc.description = "Test job submission with the aiida_crystal17 plugin"
calc.set_max_wallclock_seconds(30)
calc.set_withmpi(False)
calc.set_resources({"num_machines": 1, "num_mpiprocs_per_machine": 1})

calc.use_parameters(pdata)
calc.use_structure(newstruct)
calc.use_settings(settings)
calc.use_basisset_from_family("sto3g")

calc.store_all()

calc.submit()
```

1.4 Main Calculation Immigration

In order to immigrate existing CRYSTAL17 calculations, the *create_inputs()* function has been written to take a `.d12` and `.out` file set and create the inputs required for `crystal17.main`:

```
>>> from aiida_crystal17.tests import TEST_DIR
>>> from aiida_crystal17.parsers.migrate import create_inputs

>>> inpath = os.path.join(TEST_DIR, "input_files",
...                       'nio_sto3g_afm.crystal.d12')
>>> outpath = os.path.join(TEST_DIR, "output_files",
...                        'nio_sto3g_afm.crystal.out')

>>> inputs = create_inputs(inpath, outpath)

>>> print(inputs)
{'basis':
   {'Ni': <BasisSetData: uuid: f5edf8a7-23ca-4383-8aca-07cf22fdfbc4 (unstored)>,
    'O': <BasisSetData: uuid: 95859f1b-3822-4b60-92b2-238ec5a1931c (unstored)>},
 'parameters': <ParameterData: uuid: 16d9deb4-150a-455f-9055-cca6b1e0d93d (unstored)>,
 'structure': <StructureData: uuid: efaff664-41cc-4339-98d7-ea7594dfce52 (unstored)>,
 'settings': <ParameterData: uuid: 0ef11c68-32a5-4f5b-a783-d2e24da74328 (unstored)>}
```

This function is used by the *migrate_as_main()* function, to create a full imitation of a `crystal17.main` calculation:

```
>>> from aiida import load_dbenv
>>> load_dbenv()
>>> from aiida_crystal17.workflows.cry_main_immigrant import migrate_as_main
>>> inpath = os.path.join("input_files", 'nio_sto3g_afm.crystal.d12')
>>> outpath = os.path.join("output_files", 'nio_sto3g_afm.crystal.out')
>>> node = migrate_as_main(TEST_DIR, inpath, outpath)
>>> print(node.pk)
2474
```

In the terminal this then looks like:

```
>>> verdi calculation show 2474
-----------  ------------------------------------------------------------------------------
 ↪--------------------------------------------
type         WorkCalculation
pk           2474
uuid         b1812e1a-2576-4c70-8376-595dcde324b2
label        CryMainImmigrant
description  an immigrated CRYSTAL17 calculation into the <class 'aiida_crystal17.
 ↪calculations.cry_main.CryMainCalculation'> format
ctime        2018-09-09 00:51:27.256031+00:00
mtime        2018-09-09 00:51:28.485742+00:00
-----------  ------------------------------------------------------------------------------
 ↪--------------------------------------------
##### INPUTS:
Link label       PK  Type
------------  ----  -------------
basis_Ni       2456  BasisSetData
basis_O        2453  BasisSetData
parameters     2471  ParameterData
structure      2472  StructureData
```

(continues on next page)

(continued from previous page)

```
settings      2473  ParameterData
##### OUTPUTS:
Link label           PK  Type
-----------------  ----  -------------
output_parameters  2476  ParameterData
retrieved          2478  FolderData
```

Note: There is also a `crystal17.immigrant` calculation plugin, which works the same as PWscf immigration. However, since this approach no longer works in `aiida>=1.0`, it will be subject to change (see this ongoing issue).

CHAPTER 2

API

2.1 Subpackages

2.1.1 aiida_crystal17.calculations package

Submodules

aiida_crystal17.calculations.cry_basic module

Plugin to create a CRYSTAL17 output file from a supplied input file.

class aiida_crystal17.calculations.cry_basic.**CryBasicCalculation**(***kwargs*)
Bases: aiida.orm.implementation.django.calculation.job.JobCalculation

AiiDA calculation plugin wrapping the runcry17 executable.

aiida_crystal17.calculations.cry_main module

Plugin to create a CRYSTAL17 output file from input files created via data nodes

class aiida_crystal17.calculations.cry_main.**CryMainCalculation**(***kwargs*)
Bases: aiida.orm.implementation.django.calculation.job.JobCalculation

AiiDA calculation plugin wrapping the runcry17 executable.

classmethod get_linkname_basisset(*element*)
The name of the link used for the basis set for atomic element 'element'. It appends the basis name to the basisset_prefix, as returned by the _get_linkname_basisset_prefix() method.

Parameters **element** – a string for the atomic element for which we want to get the link name

input_schema = {u'$schema': u'http://json-schema.org/draft-04/schema#', u'additionalP

classmethod prepare_and_validate(*param_dict*, *structure*, *settings*, *basis_family=None*, *flattened=False*)
prepare and validate the inputs to the calculation

Parameters

- **input** – dict giving data to create the input .d12 file
- **structure** – the StructureData
- **settings** – StructSettingsData giving symmetry operations, etc
- **basis_family** – string of the BasisSetFamily to use
- **flattened** – whether the input dictionary is flattened

Returns parameters

settings_schema
A class that, when used as a decorator, works as if the two decorators @property and @classmethod where applied together (i.e., the object works as a property, both for the Class and for any of its instance; and is called with the class cls rather than with the instance as its first argument).

use_basisset_from_family(*family_name*)
Set the basis set to use for all atomic types, picking basis sets from the family with name family_name.

Note The structure must already be set.

Parameters **family_name** – the name of the group containing the basis sets

aiida_crystal17.calculations.cry_main_immigrant module

plugin to immigrate existing CRYSTAL17 calculation into AiiDa

class aiida_crystal17.calculations.cry_main_immigrant.**CryMainImmigrantCalculation**(***kwargs*)
Bases: *aiida_crystal17.calculations.cry_main.CryMainCalculation*

Create a CryMainCalculation object that can be used to import old jobs.

This is a sublass of aiida_crystal17.calculations.cry_main.CryMainCalculation with slight modifications to some of the class variables and additional methods that

a. parse the job's input file to create the calculation's input nodes that would exist if the calculation were submitted using AiiDa,

b. bypass the functions of the daemon, and prepare the node's attributes such that all the processes (copying of the files to the repository, results parsing, ect.) can be performed

---

Note: The keyword arguments of CryMainCalculation are also available.

Parameters

- **remote_workdir** (*str*) – Absolute path to the directory where the job was run. The transport of the computer you link ask input to the calculation is the transport that will be used to retrieve the calculation's files. Therefore, `remote_workdir` should be the absolute path to the job's directory on that computer.
- **input_file_name** (*str*) – The file name of the job's input file.
- **output_file_name** (*str*) – The file name of the job's output file (i.e. the file containing the .out of CRYSTAL17).

create_input_nodes(*open_transport*, *input_file_name=None*, *output_file_name=None*, *remote_workdir=None*)

prepare_for_retrieval_and_parsing(*open_transport*)
Tell the daemon that the calculation is computed and ready to be parsed.

Parameters **open_transport** – An open instance of the transport class of the calculation's computer. See the tutorial for more information. aiida.transport.plugins.local.LocalTransport or aiida.transport.plugins.ssh.SshTransport

The next time the daemon updates the status of calculations, it will see this job is in the 'COMPUTED' state and will retrieve its output files and parse the results.

If the daemon is not currently running, nothing will happen until it is started again.

This method also stores the calculation and all input nodes. It also copies the original input file to the calculation's repository folder.

Raises

- **aiida.common.exceptions.InputValidationError** – if `open_transport` is a different type of transport than the computer's.
- **aiida.common.exceptions.InvalidOperation** – if `open_transport` is not open.

set_input_file_name(*input_file_name*)
Set the file name of the job's input file (e.g. `'main.d12'`).

Parameters **input_file_name** (*str*) – The file name of the job's input file.

set_output_file_name(*output_file_name*)
Set the file name of the job's output file (e.g. `'pw.out'`).

Parameters **output_file_name** (*str*) – The file name of file containing the job's stdout.

set_remote_workdir(*remote_workdir*)
Set the job's remote working directory.

Parameters **remote_workdir** (*str*) – Absolute path of the job's remote working directory.

Module contents

2.1.2 aiida_crystal17.data package

Submodules

aiida_crystal17.data.basis_set module

a data type to store CRYSTAL17 basis sets

class aiida_crystal17.data.basis_set.**BasisSetData**(***kwargs*)
Bases: aiida.orm.data.Data

a data type to store CRYSTAL17 basis sets it is intended to work much like the UpfData type

- The basis file must contain one basis set in the CRYSTAL17 format
- lines beginning # will be ignored
- **the file can also start with a fenced, yaml formatted header section (starting/ending '—')**
  - Note keys should not contain '.'s

- only the actual basis data (not commented lines or the header section) will be stored as a file and hashed

Example file

```
# an ignored comment
---
author: J Smith
year: 1999
---
8 2
1 0 3  2.  0.
1 1 3  6.  0.
```

basisfamily_type_string = 'data.basisset.family'

content
return the content string for insertion into .d12 file

Returns content_str

element
return the element symbol associated with the basis set

filename
Returns the name of the file stored

classmethod from_md5(*md5*)
Return a list of all Basis Sets that match a given MD5 hash.

Note that the hash has to be stored in a _md5 attribute, otherwise the basis will not be found.

get_basis_family_names()
Get the list of all basiset family names to which the basis belongs

classmethod get_basis_group(*group_name*)
Return the BasisFamily group with the given name.

classmethod get_basis_group_map(*group_name*)
Return an {element: basis} map for the BasisFamily group with the given name.

classmethod get_basis_groups(*filter_elements=None*, *user=None*)
Return all names of groups of type BasisFamily, possibly with some filters.

Parameters

- **filter_elements** – A string or a list of strings. If present, returns only the groups that contains one Basis for every element present in the list. Default=None, meaning that all families are returned.
- **user** – if None (default), return the groups for all users. If defined, it should be either a DbUser instance, or a string for the username (that is, the user email).

get_file_abs_path()
Return the absolute path to the file in the repository

classmethod get_or_create(*filepath*, *use_first=False*, *store_basis=True*)
Pass the same parameter of the init; if a file with the same md5 is found, that BasisSetData is returned.

Parameters

- **filepath** – an absolute filename on disk
- **use_first** – if False (default), raise an exception if more than one basis set is found. If it is True, instead, use the first available basis set.

- **store_basis** (*bool*) – If false, the BasisSetData objects are not stored in the database. default=True.

Return (basis, created) where basis is the BasisSetData object, and create is either True if the object was created, or False if the object was retrieved from the DB.

md5sum
return the md5 hash of the basis set

Returns

metadata
return the attribute data as a nested dictionary

Returns metadata dict

set_file (*filepath*)
pre-parse the file to store the attributes and content separately.

store (*with_transaction=True*, *use_cache=None*)
Store a new node in the DB, also saving its repository directory and attributes, and reparsing the file so that the md5 and the element are correctly reset.

After being called attributes cannot be changed anymore! Instead, extras can be changed only AFTER calling this store() function.

Note After successful storage, those links that are in the cache, and for which also the parent node is already stored, will be automatically stored. The others will remain unstored.

Parameters **with_transaction** – if False, no transaction is used. This is meant to be used ONLY if the outer calling function has already a transaction open!

classmethod upload_basisset_family (*folder*, *group_name*, *group_description*, *stop_if_existing=True*, *extension='.basis'*, *dry_run=False*)
Upload a set of Basis Set files in a given group.

Parameters

- **folder** – a path containing all Basis Set files to be added. Only files ending in the set extension (case-insensitive) are considered.
- **group_name** – the name of the group to create. If it exists and is non-empty, a UniquenessError is raised.
- **group_description** – a string to be set as the group description. Overwrites previous descriptions, if the group was existing.
- **stop_if_existing** – if True, check for the md5 of the files and, if the file already exists in the DB, raises a MultipleObjectsError. If False, simply adds the existing BasisSetData node to the group.
- **extension** – the filename extension to look for
- **dry_run** – If True, do not change the database.

aiida_crystal17.data.basis_set.**get_basissets_by_kind** (*structure*, *family_name*)
Get a dictionary of {kind: basis} for all the kinds within the given structure using the given basis set family name.

Parameters

- **structure** – The structure that will be used.
- **family_name** – the name of the group containing the basis sets

aiida_crystal17.data.basis_set.**get_basissets_from_structure**(*structure*, *family_name*, *by_kind=False*)

Given a family name (a BasisSetFamily group in the DB) and an AiiDA structure, return a dictionary associating each element or kind name (if by_kind=True) with its BasisSetData object.

Raises

- **aiida.common.exceptions.MultipleObjectsError** – if more than one Basis Set for the same element is found in the group.
- **aiida.common.exceptions.NotExistent** – if no Basis Set for an element in the group is found in the group.

aiida_crystal17.data.basis_set.**md5_from_string**(*string*, *encoding='utf-8'*)

return md5 hash of string

Parameters

- **string** – the string to hash
- **encoding** – the encoding to use

Returns

aiida_crystal17.data.basis_set.**parse_basis**(*fname*)

get relevant information from the basis file

Parameters **fname** – the file path

Returns (metadata_dict, content_str)

- The basis file must contain one basis set in the CRYSTAL17 format
- blank lines and lines beginning '#' will be ignored
- **the file can also start with a fenced (with —), yaml formatted header section**
  - Note keys should not contain '.'s

Example

```
# an ignored comment
---
author: J Smith
year: 1999
---
8 2
1 0 3  2.  0.
1 1 3  6.  0.
```

aiida_crystal17.data.basis_set.**validate_basis_string**(*instr*)

validate that only one basis set is present, in a recognised format

Parameters **instr** – content of basis set

Returns passed

aiida_crystal17.data.struct_settings module

class aiida_crystal17.data.struct_settings.**StructSettingsData**(***kwargs*)

Bases: aiida.orm.data.Data

Stores input symmetry and kind specific setting for a structure (as required by CRYSTAL17)

- symmetry operations are stored on file (in the style of ArrayData)
- the rest of the values are stored as attributes in the database

add_path(*src_abs*, *dst_path*)
Copy a file or folder from a local file inside the repository directory. If there is a subpath, folders will be created.

Copy to a cache directory if the entry has not been saved yet.

Parameters

- **src_abs** (*str*) – the absolute path of the file to copy.
- **dst_filename** (*str*) – the (relative) path on which to copy.

Todo in the future, add an add_attachment() that has the same meaning of a extras file. Decide also how to store. If in two separate subfolders, remember to reset the limit.

compare_operations(*ops*, *decimal=5*)
compare operations against stored ones

Parameters

- **ops** – list of (flattened) symmetry operations
- **decimal** – number of decimal points to round values to

Returns dict of differences

crystal_system
get the string version of the crystal system (e.g. 'triclinic')

crystallographic_transform
get the primitive to crystallographic transformation matrix

data
Return the data as an AttributeDict

data_schema = {'$schema': 'http://json-schema.org/draft-04/schema#', 'additionalPrope

num_symops

set_data(*data*)
Replace the current data with another one.

Parameters **data** – The dictionary to set.

space_group

Module contents

Data types provided by plugin

Register data types via the "aiida.data" entry point in setup.json.

2.1.3 aiida_crystal17.parsers package

Submodules

aiida_crystal17.parsers.cry_basic module

A parser to read output from a standard CRYSTAL17 run

class aiida_crystal17.parsers.cry_basic.**CryBasicParser**(*calculation*)
Bases: aiida.parsers.parser.Parser

Parser class for parsing (stdout) output of a standard CRYSTAL17 run

check_state()
Log an error if the calculation being parsed is not in PARSING state.

get_folder(*retrieved*)
Convenient access to the retrieved folder.

classmethod get_linkname_outsettings()
Returns the name of the link to the output_structure Node exists if positions or cell changed.

classmethod get_linkname_outstructure()
Returns the name of the link to the output_structure Node exists if positions or cell changed.

parse_with_retrieved(*retrieved*)
Parse outputs, store results in database.

Parameters **retrieved** – a dictionary of retrieved nodes, where the key is the link name

Returns

a tuple with two values `(bool, node_list)`, where:

- `bool`: variable to tell if the parsing succeeded
- `node_list`: list of new nodes to be stored in the db (as a list of tuples `(link_name, node)`)

aiida_crystal17.parsers.geometry module

This module deals with reading/creating .gui files for use with the EXTERNAL keyword

File Format

```
dimesionality origin_setting crystal_type energy(optional)
    a_x a_y a_z
    b_x b_y b_z
    c_x c_y c_z
num_symm_ops (in cartesian coordinates)
    op1_rot_00 op1_rot_01 op1_rot_02
    op1_rot_10 op1_rot_11 op1_rot_12
    op1_rot_20 op1_rot_21 op1_rot_22
    op1_trans_0 op1_trans_1 op1_trans_2
    ...
num_atoms (if cryversion<17 irreducible atoms only)
    atomic_number x y z (in cartesian coordinates)
    ...
space_group_int_num num_symm_ops
```

aiida_crystal17.parsers.geometry.**ase_to_structdict**(*atoms*)
convert ase.Atoms to struct dict

Return structdict dict containing ‘lattice’, ‘atomic_numbers’, ‘pbc’, ‘ccoords’, ‘equivalent’

aiida_crystal17.parsers.geometry.**cart2frac**(*lattice*, *ccoords*)
a function that takes the cell parameters, in angstrom, and a list of Cartesian coordinates and returns the structure in fractional coordinates

aiida_crystal17.parsers.geometry.**compute_symmetry_3d**(*structdata*, *standardize*, *primitive*, *idealize*, *symprec*, *angletol*)
create 3d geometry input for CRYSTAL17

Parameters

- **structdata** – "lattice", "atomic_numbers", "ccoords", "pbc" and (optionally) "equivalent"
- **standardize** – whether to standardize the structure
- **primitive** – whether to create a primitive structure
- **idealize** – whether to idealize the structure
- **symprec** – symmetry precision to parse to spglib
- **angletol** – angletol to parse to spglib

Returns (structdata, symmdata)

aiida_crystal17.parsers.geometry.**crystal17_gui_string**(*structdata*, *symmdata*, *fractional_ops=True*)
create string of gui file content (for CRYSTAL17)

Parameters

- **structdata** – dictionary of structure data with keys: 'pbc', 'atomic_numbers', 'ccoords', 'lattice'
- **symmdata** – dictionary of symmetry data with keys: 'crystal_type', 'centring_code', 'space_group', 'operations'
- **fractional_ops** – whether the symmetry operations are in fractional coordinates

Returns

aiida_crystal17.parsers.geometry.**dict_to_structure**(*structdict*, *logger=None*)
create a dictionary of structure properties per atom

Param dictionary containing; 'lattice', 'atomic_numbers' (or 'symbols'), 'ccoords', 'pbc', 'kinds', 'equivalent'

Parameters **logger** – a logger with a *warning* method

Return structure the input structure

Rtype structure aiida.orm.data.structure.StructureData

aiida_crystal17.parsers.geometry.**frac2cart**(*lattice*, *fcoords*)
a function that takes the cell parameters, in angstrom, and a list of fractional coordinates and returns the structure in cartesian coordinates

aiida_crystal17.parsers.geometry.**get_centering_code**(*sg_number*, *sg_symbol*)
get crystal centering codes, to convert from primitive to conventional

Parameters

- **sg_number** – the space group number
- **sg_symbol** – the space group symbol

Returns CRYSTAL centering code

aiida_crystal17.parsers.geometry.**get_crystal_system**(*sg_number*, *as_number=False*)
Get the crystal system for the structure, e.g., (triclinic, orthorhombic, cubic, etc.) from the space group number

Parameters

- **sg_number** – the spacegroup number
- **as_number** – return the system as a number (recognized by CRYSTAL) or a str

Returns Crystal system for structure or None if system cannot be detected.

aiida_crystal17.parsers.geometry.**get_lattice_type**(*sg_number*)
Get the lattice for the structure, e.g., (triclinic, orthorhombic, cubic, etc.).This is the same than the crystal system with the exception of the hexagonal/rhombohedral lattice

Parameters **sg_number** – space group number

Returns Lattice type for structure or None if type cannot be detected.

aiida_crystal17.parsers.geometry.**ops_cart_to_frac**(*ops_flat*, *lattice*)
convert a list of flattened cartesian symmetry operations to fractional

aiida_crystal17.parsers.geometry.**ops_frac_to_cart**(*ops_flat*, *lattice*)
convert a list of flattened fractional symmetry operations to cartesian

aiida_crystal17.parsers.geometry.**read_gui_file**(*fpath*, *cryversion=17*)
read CRYSTAL geometry (.gui) file

Parameters

- **fpath** (*str or pathlib.Path*) – path to file
- **cryversion** (*int*) – version of CRYSTAL

Returns

aiida_crystal17.parsers.geometry.**structdict_to_ase**(*structdict*)
convert struct dict to ase.Atoms

Parameters **structdict** – dict containing 'lattice', 'atomic_numbers', 'pbc', 'ccoords', 'equivalent'

Return type ase.Atoms

aiida_crystal17.parsers.geometry.**structure_to_dict**(*structure*)
create a dictionary of structure properties per atom

Parameters **structure** (*aiida.orm.data.structure.StructureData*) – the input structure

Returns dictionary containing; lattice, atomic_numbers, ccoords, pbc, kinds, equivalent

Return type dict

aiida_crystal17.parsers.inputd12_read module

module for reading main.d12 (for immigration)

aiida_crystal17.parsers.inputd12_read.**extract_data**(*input_string*)
extract data from a main.d12 CRYSTAL17 file

- Any geometry creation commands are ignored

- Basis sets must be included explicitly (no keywords) and are read into the basis_sets list
- FRAGMENT, GHOSTS and ATOMSPIN commands are read into the atom_props dict
- Otherwise, only commands contained in the inputd12.schema.json are allowed

Parameters **input_string** – a string if the content of the file

Returns output_dict the paramtere dict for use in `crystal17.main` calculation

Returns basis_sets a list of the basis sets

Returns atom_props a dictionary of atom specific values (spin_alpha, spin_beta, ghosts, fragment)

aiida_crystal17.parsers.inputd12_write module

module to write CRYSTAL17 .d12 files

aiida_crystal17.parsers.inputd12_write.**format_value**(*dct*, *keys*)
return the value + a new line, or empty string if keys not found

aiida_crystal17.parsers.inputd12_write.**write_input**(*indict*, *basis_sets*, *atom_props=None*)
write input of a validated input dictionary

Parameters

- **indict** – dictionary of input
- **basis_sets** – list of basis set strings or objects with *content* property
- **atom_props** – dictionary of atom ids with specific properties ("spin_alpha", "spin_beta", "unfixed", "ghosts")

Returns

aiida_crystal17.parsers.mainout_parse module

parse the main output file and create the required output nodes

aiida_crystal17.parsers.mainout_parse.**parse_mainout**(*abs_path*, *parser_class*, *init_struct=None*, *init_settings=None*)
parse the main output file and create the required output nodes

Parameters

- **abs_path** – absolute path of stdout file
- **parser_class** – a string denoting the parser class
- **init_struct** – input structure
- **init_settings** – input structure settings

Return psuccess a boolean that is False in case of failed calculations

Return output_nodes containing "paramaters" and (optionally) "structure" and "settings"

aiida_crystal17.parsers.migrate module

module to create inputs from existing CRYSTAL17 runs

aiida_crystal17.parsers.migrate.**create_inputs**(*inpath*, *outpath*)
create `crystal17.main` input nodes from an existing run

NB: none of the nodes are stored, also existing basis will be retrieved if availiable

Parameters

- **inpath** – path to .d12 file
- **outpath** – path to .out file

Returns dictionary of inputs, with keys 'structure', 'parameters', 'settings', 'structure', 'basis'

Module contents

parsers for CRYSTAL17

2.1.4 aiida_crystal17.workflows package

Submodules

aiida_crystal17.workflows.symmetrise_3d_struct module

a work flow to symmetrise a structure and compute the symmetry operations

class aiida_crystal17.workflows.symmetrise_3d_struct.**Symmetrise3DStructure**
Bases: `aiida.work.workchain.WorkChain`

modify an AiiDa structure instance and compute its symmetry, given a settings dictionary

Symmetry is restricted by atom kinds

compute()

classmethod define(*spec*)

validate()

aiida_crystal17.workflows.symmetrise_3d_struct.**run_symmetrise_3d_structure**(*structure*, *settings=None*)
run the Symmetrise3DStructure workchain and return the structure and settings data nodes, for inputting into `crystal17.main` calculation

Parameters

- **structure** – StructureData
- **settings** – dict or ParameterData

Returns (StructureData, StructSettingsData)

aiida_crystal17.workflows.cry_main_immigrant module

a workflow to immigrate previously run CRYSTAL17 computations into Aiida

class aiida_crystal17.workflows.cry_main_immigrant.**CryMainImmigrant**
Bases: aiida.work.workchain.WorkChain

an immigrant calculation of CryMainCalculation

aiida_crystal17.workflows.cry_main_immigrant.**migrate_as_main**(*work_dir*, *input_rel_path*, *output_rel_path*, *resources=None*, *input_links=None*)
migrate existing CRYSTAL17 calculation as a WorkCalculation, which imitates a crystal17.main calculation

Parameters

- **work_dir** – the absolute path to the directory to holding the files
- **input_rel_path** – relative path (from work_dir) to .d12 file
- **output_rel_path** – relative path (from work_dir) to .out file
- **resources** – a dict of of job resource parameters (not yet implemented)
- **input_links** – a dict of existing nodes to link inputs to (allowed keys: 'structure', 'settings', 'parameters')

Example of input_links={'structure': {"cif_file": CifNode}}, will create a link (via a workcalculation) from the CifNode to the input StructureData

Raises

- **IOError** – if the work_dir or files do not exist
- **aiida.common.exceptions.ParsingError** – if the input parsing fails
- **aiida.parsers.exceptions.OutputParsingError** – if the output parsing fails

Returns the calculation node

Return type aiida.orm.WorkCalculation

Module contents

2.1.5 aiida_crystal17.validation package

Module contents

aiida_crystal17.validation.**read_schema**(*name='inputd12'*)
read and return an json schema

Returns

aiida_crystal17.validation.**validate_with_dict**(*data*, *schema*)
validate json-type data against a schema

Parameters

- **data** – dictionary

- **schema** – dictionary

aiida_crystal17.validation.**validate_with_json**(*data*, *name='inputd12'*)
validate json-type data against a schema

Parameters **data** – dictionary

2.1.6 aiida_crystal17.cmndline package

Submodules

aiida_crystal17.cmndline.basis_set module

aiida_crystal17.cmndline.basis_set.**try_grab_description**(*ctx*, *param*, *value*)
Try to get the description from an existing group if it's not given.

This is a click parameter callback.

aiida_crystal17.cmndline.options module

Common click options for verdi commands

class aiida_crystal17.cmndline.options.**OverridableOption**(**args*, ***kwargs*)
Bases: object

Wrapper around click option that increases reusability

Click options are reusable already but sometimes it can improve the user interface to for example customize a help message for an option on a per-command basis. Sometimes the option should be prompted for if it is not given. On some commands an option might take any folder path, while on another the path only has to exist.

Overridable options store the arguments to click.option and only instanciate the click.Option on call, kwargs given to __call__ override the stored ones.

Example:

```
FOLDER = OverridableOption('--folder', type=click.Path(file_okay=False), help='A␣
↪folder')

@click.command()
@FOLDER(help='A folder, will be created if it does not exist')
def ls_or_create(folder):
    click.echo(os.listdir(folder))

@click.command()
@FOLDER(help='An existing folder', type=click.Path(exists=True, file_okay=False,␣
↪readable=True)
def ls(folder)
    click.echo(os.listdir(folder))
```

aiida_crystal17.cmndline.structsettings module

Module contents

2.2 Submodules

2.2.1 aiida_crystal17.utils module

common utilities

class aiida_crystal17.utils.**HelpDict**(**args*, ***kwargs*)
Bases: _abcoll.MutableMapping

a dictionary which associates help text with each key

copy()

help

aiida_crystal17.utils.**flatten_dict**(*indict*, *delimiter='.'*)

aiida_crystal17.utils.**get_keys**(*dct*, *keys*, *default=None*, *raise_error=False*)
retrieve the leaf of a key path from a dictionary

Parameters

- **dct** – the dict to search
- **keys** – key path
- **default** – default value to return
- **raise_error** – whether to raise an error if the path isn't found

Returns

aiida_crystal17.utils.**unflatten_dict**(*indict*, *delimiter='.'*)

2.2.2 aiida_crystal17.aiida_compatability module

Utilities for working with different versions of aiida

aiida_crystal17.aiida_compatability.**aiida_version**()
get the version of aiida in use

Returns packaging.version.Version

aiida_crystal17.aiida_compatability.**cmp_load_verdi_data**()
Load the verdi data click command group for any version since 0.11.

aiida_crystal17.aiida_compatability.**cmp_version**(*string*)
convert a version string to a packaging.version.Version

aiida_crystal17.aiida_compatability.**dbenv**(*function*)
A function decorator that loads the dbenv if necessary before running the function.

aiida_crystal17.aiida_compatability.**get_automatic_user**(**args*, ***kwargs*)

aiida_crystal17.aiida_compatability.**get_basic_data_pre_1_0**(**args*, ***kwargs*)

aiida_crystal17.aiida_compatability.**get_calc_log**(*calcnode*)
get a formatted string of the calculation log

aiida_crystal17.aiida_compatability.**get_data_class**(**args*, ***kwargs*)
Provide access to the orm.data classes with deferred dbenv loading.

compatiblity: also provide access to the orm.data.base memebers, which are loadable through the DataFactory as of 1.0.0-alpha only.

aiida_crystal17.aiida_compatability.**get_data_node**(*data_type*, **args*, ***kwargs*)

aiida_crystal17.aiida_compatability.**json_default**(*o*)

aiida_crystal17.aiida_compatability.**load_dbenv_if_not_loaded**(***kwargs*)
Load dbenv if necessary, run spinner meanwhile to show command hasn't crashed.

aiida_crystal17.aiida_compatability.**run_get_node**(*process*, *inputs_dict*)
an implementation of run_get_node, which is compatible with both aiida v0.12 and v1.0.0

it will also convert "options" "label" and "description" to/from the _ variant

Parameters

- **process** – a process
- **inputs_dict** (*dict*) – a dictionary of inputs

Returns the calculation Node

2.3 Module contents

aiida_crystal17

AiiDA plugin for running the CRYSTAL17 code

If you use this plugin for your research, please cite the Github repository (paper to come).

If you use AiiDA for your research, please cite the following work:

> Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky, *AiiDA: automated interactive infrastructure and database for computational science*, Comp. Mat. Sci 111, 218-230 (2016); http://dx.doi.org/10.1016/j.commatsci.2015.09.013; http://www.aiida.net.

`aiida-crystal17` is released under the MIT license.

Please contact chrisj_sewell@hotmail.com for information concerning `aiida-crystal17` and the AiiDA mailing list for questions concerning `aiida`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

D

E

F

G

H

V

W