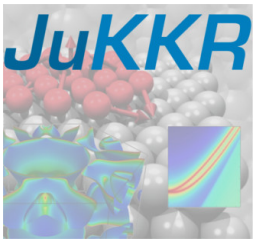
AiiDA-KKR documentation

Release 1.1.11-dev4

The AiiDA-KKR team.

Jan 15, 2021

Contents


JuKKR


AiiDA

CHAPTER 1

Welcome to documentation of the AiiDA plugin for the Jülich KKRcode!

The plugin is available at https://github.com/JuDFTteam/aiida-kkr

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

> Philipp Rüßmann, Fabian Bertoldo, and Stefan Blügel, *The AiiDA-KKR plugin and its application to high-throughput impurity embedding into a topological insulator*, arXiv:2003.08315 [cond-mat.mtrl-sci] (2020); https://arxiv.org/abs/2003.08315

Also please cite the original AiiDA paper:

> 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.

1.1 Requirements

- Installation of aiida-core
- Installation of KKR codes (*kkrhost*, *kkrimp*, *voronoi*) of the JuKKR package
- Installation of aiida-kkr

Once all requirements are installed you need to set up the computers and codes before you can submit KKR calcutions using the *aiida-kkr* plugin.

1.1.1 User's guide

1.1.1.1 User's guide

Calculations

Here the calculations of the aiida-kkr plugin are presented. It is assumed that the user already has basic knowledge of python, aiida (e.g. database structure, verdi commands, structure nodes) and KKR (e.g. LMAX cutoff, energy contour

integration). Also *aiida-kkr* should be installed as well as the Voronoi, KKR and KKRimp codes should already be configured.

In practice, the use of the workflows is more convenient but here the most basic calculations which are used underneath in the workflows are introduced step by step.

In the following the calculation plugins provided by aiida-kkr are introduced at the example of bulk Cu.

Note: If you follow the steps described here please make sure that your python script contains:

```
from aiida import load_profile
load_profile()
```

To ensure that the aiida database is properly integrated.

Voronoi starting potential generator

The Voronoi code creates starting potentials for a KKR calculation and sets up the atom-centered division of space into voronoi cells. Also corresponding shape functions are created, which are needed for full-potential corrections.

The voronoi plugin is called `kkr.voro` and it has the following input and output nodes:

Three input nodes:

- `parameters` KKR parameter set for Voronoi calculation (Dict)
- `structure` structure data node node describing the crystal lattice (StructureData)
- `code` Voronoi code node (code)

Three output nodes:

- `remote_folder` (RemoteData)
- `retrieved` (FolderData)
- `output_parameters` (Dict)

Additional optional input nodes that trigger special behavior of a Voronoi calculation are:

- `parent_KKR` (RemoteData of a KKR Calculation)
- `potential_overwrite` (SingleFileData)

Now the basic usage of the voronoi plugin is demonstrated at the example of Cu bulk for which first the aiida structure node and the parameter node containing KKR specific parameters (LMAX cutoff etc.) are created before a voronoi calculation is set up and submitted.

Input structure node

First we create an aiida structure:

```
# get aiida StructureData class:
from aiida.plugins import DataFactory
StructureData = DataFactory('structure')
```

Then we create the aiida StructureData node (here for bulk Cu):

```
alat = 3.61 # lattice constant in Angstroem
bravais = [[0.5*alat, 0.5*alat, 0], [0.5*alat, 0, 0.5*alat], [0, 0.5*alat, 0.5*alat]]
 ↪# Bravais matrix in Ang. units
# now create StructureData instance and set Bravais matrix and atom in unit cell
Cu = StructureData(cell=bravais)
Cu.append_atom(position=[0,0,0], symbols='Cu')
```

Input parameter node

Next we create an empty set of KKR parameters (LMAX cutoff etc.) for voronoi code:

```
# load kkrparms class which is a useful tool to create the set of input parameters␣
 ↪for KKR-family of calculations
from masci_tools.io.kkr_params import kkrparams
params = kkrparams(params_type='voronoi')
```

Note: we can find out which parameters are mandatory to be set using `missing_params = params.get_missing_keys(use_aiida=True)`

and set at least the mandatory parameters:

```
params.set_multiple_values(LMAX=2, NSPIN=1, RCLUSTZ=2.3)
```

finally create an aiida Dict node and fill with the dictionary of parameters:

```
Dict = DataFactory('dict') # use DataFactory to get ParamerterData class
ParaNode = Dict(dict=params.get_dict())
```

Submit calculation

Now we get the voronoi code:

```
from aiida.orm import Code # load aiida 'Code' class

codename = 'voronoi@localhost'
code = Code.get_from_string(codename)
```

Note: Make sure that the voronoi code is installed: `verdi code list` should give you a list of installed codes where *codename* should be in.

and create new process builder for a VoronoiCalculation:

```
builder = code.get_builder()
```

Note: This will already set `builder.code` to the voronoi code which we loaded above.

and set resources that will be used (here serial job) in the options dict of the metadata:

```
builder.metadata.options = {'resources': {'num_machines':1, 'tot_num_mpiprocs':1} }
```

Note: If you use a computer without a default queue you need to set the name of the queue as well: `builder.metadata.options['queue_name'] = 'th1'`)

then set structure and input parameter:

```
builder.structure = Cu
builder.parameters = ParaNode
```

Note: Additionally you could set the `parent_KKR` and `potential_overwrite` input nodes which trigger special run modes of the voronoi code that are discussed below.

Now we are ready to submit the calculation:

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

Note: check calculation state (or use verdi calculation list -a -p1) using `voro_calc.process_state`

Voronoi calculation with the `parent_KKR` input node

To come . . .

Voronoi calculation with the `potential_overwrite` input node

To come . . .

KKR calculation for bulk and interfaces

A KKR calculation is provided by the `kkr.kkr` plugin, which has the following input and output nodes.

Three input nodes:

- `parameters` KKR parameter fitting the requirements for a KKR calculation (Dict)
- `parent_folder` parent calulation remote folder node (RemoteFolder)
- `code` KKR code node (code)

Three output nodes:

- `remote_folder` (RemoteData)
- `retrieved` (FolderData)
- `output_parameters` (Dict)

Note: The parent calculation can be one of the following:

1. Voronoi calculation, initial calculation starting from structure
2. previous KKR calculation, e.g. preconverged calculation

The necessary structure information is always extracted from the voronoi parent calculation. In case of a continued calculation the voronoi parent is recuresively searched for.

Special features exist where a fourth input node is persent and which triggers special behavior of the KKR calculation:

- `impurity_info` Node specifying the impurity cluster (*Dict*)
- `kpoints` Node specifying the kpoints for which the bandstructure is supposed to be calculated (*Kpoints-Data*)

The different possible modes to run a kkr calculation (start from Voronoi calculation, continue from previous KKR calculation, *host Greenfunction writeout* feature) are demonstrated in the following.

Start KKR calculation from voronoi parent

Reuse settings from voronoi calculation:

```
voronoi_calc_folder = voro_calc.out.remote_folder
voro_params = voro_calc.inputs.parameters
```

Now we update the KKR parameter set to meet the requirements for a KKR calculation (slightly different than voronoi calculation). Thus, we create a new set of parameters for a KKR calculation and fill the already set values from the previous voronoin calculation:

```
# new kkrparams instance for KKR calculation
params = kkrparams(params_type='kkr', **voro_params.get_dict())

# set the missing values
params.set_multiple_values(RMAX=7., GMAX=65.)

# choose 20 simple mixing iterations first to preconverge potential (here 5% simple␣
↪mixing)
params.set_multiple_values(NSTEPS=20, IMIX=0, STRMIX=0.05)

# create aiida Dict node from the KKR parameters
ParaNode = Dict(dict=params.get_dict())
```

Note: You can find out which parameters are missing for the KKR calculation using `params.get_missing_keys()`

Now we can get the KKR code and create a new calculation instance and set the input nodes accordingly:

```
code = Code.get_from_string('KKRcode@localhost')
builder = code.get_builder()

# set input Parameter, parent calulation (previous voronoi calculation), computer␣
↪resources
builder.parameters = ParaNode
builder.parent_folder = voronoi_calc_folder
builder.metadata.options = {'resources' :{'num_machines': 1, 'num_mpiprocs_per_machine
↪':1}}
```

We can then run the KKR calculation:

```
kkr_calc = submit(builder)
```

Continue KKR calculation from KKR parent calculation

First we create a new KKR calculation instance to continue KKR ontop of a previous KKR calclation:

```
builder = code.get_builder()
```

Next we reuse the old KKR parameters and update scf settings (default is NSTEPS=1, IMIX=0):

```
params.set_multiple_values(NSTEPS=50, IMIX=5)
```

and create the aiida Dict node:

```
ParaNode = Dict(dict=params.get_dict())
```

Then we set the input nodes for calculation:

```
builder.parameters = ParaNode
kkr_calc_parent_folder = kkr_calc.outputs.remote_folder # parent remote folder of␣
↪previous calculation
builder.parent_folder = kkr_calc_parent_folder
builder.metadata.options = {'resources': {'num_machines': 1, 'num_mpiprocs_per_machine
↪':1}}
```

store input nodes and submit calculation:

```
kkr_calc_continued = submit(builder)
```

The finished calculation should have this output node that can be access within python using `kkr_calc_continued.outputs.output_parameters.get_dict()`. An excerpt of the ouput dictionary may look like this:

```
{u'alat_internal': 4.82381975,
 u'alat_internal_unit': u'a_Bohr',
 u'convergence_group': {
     u'calculation_converged': True,
     u'charge_neutrality': -1.1e-05,
     u'nsteps_exhausted': False,
     u'number_of_iterations': 47,
     u'rms': 6.4012e-08,
     ...},
u'energy': -44965.5181266111,
u'energy_unit': u'eV',
u'fermi_energy': 0.6285993399,
u'fermi_energy_units': u'Ry',
u'nspin': 1,
u'number_of_atoms_in_unit_cell': 1,
u'parser_errors': [],
...
u'warnings_group': {u'number_of_warnings': 0, u'warnings_list': []}}
```

Special run modes: host GF writeout (for KKRimp)

Here we take the remote folder of the converged calculation to reuse settings and write out Green function and tmat of the crystalline host system:

```
kkr_converged_parent_folder = kkr_calc_continued.outputs.remote_folder
```

Now we extract the parameters of the kkr calculation and add the `KKRFLEX` run-option:

```
kkrcalc_converged = kkr_converged_parent_folder.get_incoming().first().node
kkr_params_dict = kkrcalc_converged.inputs.parameters.get_dict()
kkr_params_dict['RUNOPT'] = ['KKRFLEX']
```

The parameters dictionary is not passed to the aiida Dict node:

```
ParaNode = Dict(dict=kkr_params_dict)
```

Now we create a new KKR calculation and set input nodes:

```
code = kkrcalc_converged.inputs.code # take the same code as in the calculation before
builder= code.get_builder()
resources = kkrcalc_converged.attributes['resources']
builder.metadata.options = {'resources': resources}
builder.parameters = ParaNode
builder.parent_folder = kkr_converged_parent_folder
# prepare impurity_info node containing the information about the impurity cluster
imp_info = Dict(dict={'Rcut':1.01, 'ilayer_center': 0, 'Zimp':[79.]})
# set impurity info node to calculation
builder.impurity_info = imp_info
```

Note: The `impurity_info` node should be a Dict node and its dictionary should describe the impurity cluster using the following parameters:

- `ilayer_center` (int) layer index of position in the unit cell that describes the center of the impurity cluster
- `Rcut` (float) cluster radius of impurity cluster in units of the lattice constant
- `hcut` (float, *optional*) height of a cylindrical cluster with radius `Rcut`, if not given spherical cluster is taken
- `cylinder_orient` (list of 3 float values, *optional*)
- `Zimp` (list of *Nimp* float entries) atomic charges of the substitutional impurities on positions defined by `Rimp_rel`
- `Rimp_rel` (list of *Nimp* [float, float, float] entries, *optional*, defaults to [0,0,0] for single impurity) cartesian positions of all *Nimp* impurities, relative to the center of cluster (i.e. position defined by `ilayer_center`)
- `imp_cls` (list of [float, float, float, int] entries, *optional*) full list of impurity cluster positions and layer indices *(x, y, z, ilayer)*, overwrites auto generation using `Rcut` and `hcut` settings

> **Warning:** `imp_cls` functionality not implemented yet

The calculation can then be submitted:

```
# submit calculation
GF_host_calc = submit(builder)
```

Once the calculation has finished the retrieve folder should contain the `kkrflex_*` files needed for the impurity calculation.

Special run modes: bandstructure

Here we take the remote folder of the converged calculation and compute the bandstructure of the Cu bulk system. We reuse the DOS settings for the energy interval in which the bandstructure is computed from a previous calculation:

```
from aiida.orm import load_node
kkr_calc_converged = load_node(<-id-of-previous-calc>)
kkr_dos_calc = load_node(<-id-of-previous-DOS-calc>)
```

Now we need to generate the kpoints node for bandstructure calculation. This is done using aiida's `get_explicit_kpoints_path` function that extracts the kpoints along high symmetry lines from a structure:

```
# first extract the structure node from the KKR parent calculation
from aiida_kkr.calculations.voro import VoronoiCalculation
struc, voro_parent = VoronoiCalculation.find_parent_structure(kkr_calc_converged.
↪outputs.remote_folder)
# then create KpointsData node
from aiida.tools.data.array.kpoints import get_explicit_kpoints_path
kpts = get_explicit_kpoints_path(struc).get('explicit_kpoints')
```

Warning: Note that the `get_explicit_kpoints_path` function returns kpoints for the primitive structure. In this example the input structure is already the primitive cell however in general this may not always be the case.

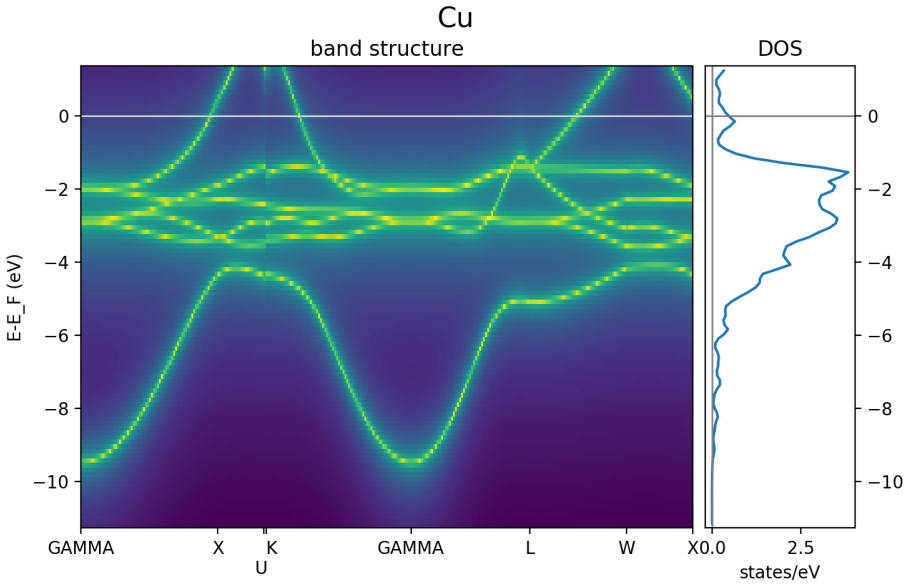
Then we set the `kpoints` input node to a new KKR calculation and change some settings of the input parameters accordingly (i.e. energy contour like in DOS run):

```
# create bandstructure calculation reusing old settings (including same computer and␣
↪resources in this example)
kkrcode = kkr_calc_converged.inputs.code
builder = kkrcode.get_builder()
builder.kpoints = kpts # pass kpoints as input
builder.parent_folder = kkr_calc_converged.outputs.remote_folder
builder.metadata.options = {'resources': kkr_calc_converged.attributes['resources']}
# change parameters to qdos settings (E range and number of points)
from masci_tools.io.kkr_params import kkrparams
qdos_params = kkrparams(**kkr_calc_converged.inputs.parameters.get_dict()) # reuse␣
↪old settings
# reuse the same emin/emax settings as in DOS run (extracted from input parameter␣
↪node)
qdos_params.set_multiple_values(EMIN=host_dos_calc.inputs.parameters.get_dict().get(
↪'EMIN'),
                                EMAX=host_dos_calc.inputs.parameters.get_dict().get(
↪'EMAX'),
                                NPT2=100)
builder.parameters = Dict(dict=qdos_params.get_dict())
```

The calculation is then ready to be submitted:

```
# submit calculation
kkrcalc = submit(builder)
```

The result of the calculation will then contain the `qdos.aa.s.dat` files in the retrieved node, where `aa` is the atom index and `s` the spin index of all atoms in the unit cell. The resulting bandstructure (for the Cu bulk test system considered here) should look like this (see *here for the plotting script*):



Special run modes: Jij extraction

The extraction of exchange coupling parameters is triggered with the `XCPL` run option and needs at lest the `JIJRAD` paramter to be set. Here we take the remote folder of the converged calculation and compute the exchange parameters:

```
from aiida.orm import load_node
kkr_calc_converged = load_node(<-id-of-previous-calc>)
```

Then we set the `XCLP` run option and the `JIJRAD` parameter (the `JIJRADXY`, `JIJSITEI` and `JIJSITEJ` parameters are not mandatory and are ommitted in this example) in the input node to a new KKR calculation:

```
# create bandstructure calculation reusing old settings (including same computer and␣
↪resources in this example)
kkrcode = kkr_calc_converged.inputs.code
builder = kkrcode.get_builder()
builder.parent_folder = kkr_calc_converged.outputs.remote_folder
builder.metadata.options = {'resources': kkr_calc_converged.attributes['resources']}
# change parameters to Jij settings ('XCPL' runopt and JIJRAD parameter)
from aiida_kkr.tools.kkr_params import kkrparams
Jij_params = kkrparams(**kkr_calc_converged.inputs.parameters.get_dict()) # reuse old␣
↪settings
# add JIJRAD (remember: in alat units)
Jij_params.set_value('JIJRAD', 1.5)
# add 'XCPL' runopt to list of runopts
runopts = Jij_params.get_value('RUNOPT')
runopts.append('XCPL    ')
Jij_params.set_value('RUNOPT', runopts)
```

(continues on next page)

(continued from previous page)

```
# now use updated parameters
builder.parameters = Dict(dict=qdos_params.get_dict())
```

The calculation is then ready to be submitted:

```
# submit calculation
kkrcalc = submit(builder)
```

The result of the calculation will then contain the `Jijatom.*` files in the retrieved node and the `shells.dat` files which allows to map the values of the exchange interaction to equivalent positions in the different shells.

KKR impurity calculation

Plugin: `kkr.kkrimp`

Four input nodes:

- `parameters`, optional: KKR parameter fitting the requirements for a KKRimp calculation (Dict)
- Only one of
  1. `impurity_potential`: starting potential for the impurity run (SingleFileData)
  2. `parent_folder`: previous KKRimp parent calulation folder (RemoteFolder)
- `code`: KKRimp code node (code)
- `host_Greenfunction_folder`: KKR parent calulation folder containing the writeout of the *host's Green function files* (RemoteFolder)

Note: If no `parameters` node is given then the default values are extracted from the `host_Greenfunction` calculation.

Three output nodes:

- `remote_folder` (RemoteData)
- `retrieved` (FolderData)
- `output_parameters` (Dict)

Note: The parent calculation can be one of the following:

1. Voronoi calculation, initial calculation starting from structure
2. previous KKR calculation, e.g. preconverged calculation

The necessary structure information is always extracted from the voronoi parent calculation. In case of a continued calculation the voronoi parent is recuresively searched for.

Create impurity potential

Now the starting potential for the impurity calculation needs to be generated. This means that we need to create an auxiliary structure which contians the impurity in the system where we want to embed it. Then we run a Voronoi calculation to create the starting potential. Here we use the example of a Au impurity embedded into bulk Cu.

The impurity code expects an aiida SingleFileData object that contains the impurity potential. This is finally constructed using the neworder_potential_wf workfunction from aiida_kkr.tools.common_workfunctions.

We start with the creation of the auxiliary styructure:

```
# use an aiida workfunction to keep track of the provenance
from aiida.work import workfunction as wf
@wf
def change_struc_imp_aux_wf(struc, imp_info): # Note: works for single imp at center␣
↪only!
    from aiida.common.constants import elements as PeriodicTableElements
    _atomic_numbers = {data['symbol']: num for num, data in PeriodicTableElements.
↪iteritems()}

    new_struc = StructureData(cell=struc.cell)
    isite = 0
    for site in struc.sites:
        sname = site.kind_name
        kind = struc.get_kind(sname)
        pos = site.position
        zatom = _atomic_numbers[kind.get_symbols_string()]
        if isite == imp_info.get_dict().get('ilayer_center'):
            zatom = imp_info.get_dict().get('Zimp')[0]
        symbol = PeriodicTableElements.get(zatom).get('symbol')
        new_struc.append_atom(position=pos, symbols=symbol)
        isite += 1

    return new_struc

new_struc = change_struc_imp_aux_wf(voro_calc.inputs.structure, imp_info)
```

Note: This functionality is alreadyincorporated in the kkr_imp_wc workflow.

Then we run the Voronoi calculation for auxiliary structure to create the impurity starting potential:

```
codename = 'voronoi@localhost'
code = Code.get_from_string(codename)

builder = code.get_builder()
builder.metadata.options = {'resources': {'num_machines':1, 'tot_num_mpiprocs':1}}
builder.structure = new_struc
builder.parameters = kkrcalc_converged.inputs.parameters

voro_calc_aux = submit(builder)
```

Now we create the impurity starting potential using the converged host potential for the surrounding of the impurity and the new Au impurity startpot:

```
from aiida_kkr.tools.common_workfunctions import neworder_potential_wf

potname_converged = kkrcalc_converged._POTENTIAL
potname_imp = 'potential_imp'
neworder_pot1 = [int(i) for i in loadtxt(GF_host_calc.outputs.retrieved.get_abs_path(
↪'scoef'), skiprows=1)[:,3]-1]
potname_impvorostart = voro_calc_aux._OUT_POTENTIAL_voronoi
```

(continues on next page)

(continued from previous page)

```
replacelist_pot2 = [[0,0]]

settings_dict = {'pot1': potname_converged,  'out_pot': potname_imp, 'neworder':␣
↪neworder_pot1,
                 'pot2': potname_impvorostart, 'replace_newpos': replacelist_pot2,
↪'label': 'startpot_KKRimp',
                 'description': 'starting potential for Au impurity in bulk Cu'}
settings = Dict(dict=settings_dict)

startpot_Au_imp_sfd = neworder_potential_wf(settings_node=settings,
                                            parent_calc_folder=kkrcalc_converged.
↪outputs.remote_folder,
                                            parent_calc_folder2=voro_calc_aux.outputs.
↪remote_folder)
```

Create and submit initial KKRimp calculation

Now we create a new impurity calculation, set all input nodes and submit the calculation to preconverge the impurity potential (Au embedded into Cu ulk host as described in the `impurity_info` node):

```
# needed to link to host GF writeout calculation
GF_host_output_folder = GF_host_calc.outputs.remote_folder

# create new KKRimp calculation
from aiida_kkr.calculations.kkrimp import KkrimpCalculation
kkrimp_calc = KkrimpCalculation()

builder = Code.get_from_string('KKRimp@my_mac')

builder.code(kkrimp_code)
builder.host_Greenfunction_folder = GF_host_output_folder
builder.impurity_potential = startpot_Au_imp_sfd
builder.resources = resources

# first set 20 simple mixing steps
kkrimp_params = kkrparams(params_type='kkrimp')
kkrimp_params.set_multiple_values(SCFSTEPS=20, IMIX=0, MIXFAC=0.05)
ParamsKKRimp = Dict(dict=kkrimp_params.get_dict())
bilder.parameters = ParamsKKRimp

# submit calculation
kkrimp_calc = submit(builder)
```

Restart KKRimp calculation from KKRimp parent

Here we demonstrate how to restart a KKRimp calculation from a parent calculation from which the starting potential is extracted autimatically. This is used to compute the converged impurity potential starting from the previous preconvergence step:

```
builder = kkrimp_code.get_builder()
builder.parent_calc_folder = kkrimp_calc.outputs.remote_folder
builder.metadata.options = {'resources': resources}
builder.host_Greenfunction_folder = kkrimp_calc.inputs.GFhost_folder
```

(continues on next page)

(continued from previous page)

```
kkrimp_params = kkrparams(params_type='kkrimp', **kkrimp_calc.inputs.parameters.get_
↪dict())
kkrimp_params.set_multiple_values(SCFSTEPS=99, IMIX=5, MIXFAC=0.05)
ParamsKKRimp = Dict(dict=kkrimp_params.get_dict())
builder.parameters = ParamsKKRimp

# submit
kkrimp_calc_converge = submit(builder)
```

Impurity DOS

create final imp DOS (new host GF for DOS contour, then KKRimp calc using converged potential)

first prepare host GF with DOS contour:

```
params = kkrparams(**GF_host_calc.inputs.parameters.get_dict())
params.set_multiple_values(EMIN=-0.2, EMAX=GF_host_calc.res.fermi_energy+0.1, NPOL=0,␣
↪NPT1=0, NPT2=101, NPT3=0)
ParaNode = Dict(dict=params.get_dict())

code = GF_host_calc.inputs.code # take the same code as in the calculation before
builder= code.new_calc()
resources = GF_host_calc.get_resources()
builder.resources = resources
builder.parameters = ParaNode
builder.parent_folder = kkr_converged_parent_folder
builder.impurity_info = GF_host_calc.inputs.impurity_info

GF_host_doscalc = submit(builder)
```

Then we run the KKRimp step using the converged potential (via the `parent_calc_folder` node) and the host GF which contains the DOS contour information (via `host_Greenfunction_folder`):

```
builder = kkrimp_calc_converge.inputs.code.get_builder()
builder.host_Greenfunction_folder(GF_host_doscalc.outputs.remote_folder)
builder.parent_calc_folder(kkrimp_calc_converge.outputs.remote_folder)
builder.resources(kkrimp_calc_converge.get_resources())

params = kkrparams(params_type='kkrimp', **kkrimp_calc_converge.inputs.parameters.get_
↪dict())
params.set_multiple_values(RUNFLAG=['lmdos'], SCFSTEPS=1)
ParaNode = Dict(dict=params.get_dict())

builder.parameters(ParaNode)

kkrimp_doscalc = submit(builder)
```

Finally we plot the DOS:

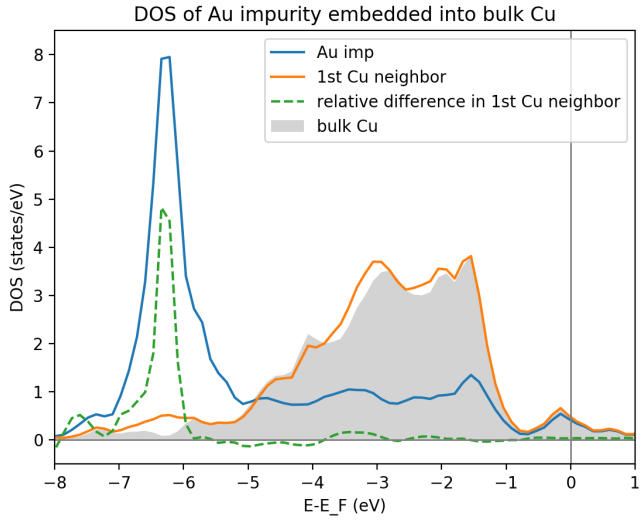
```
# get interpolated DOS from GF_host_doscalc calculation:
from masci_tools.io.common_functions import interpolate_dos
dospath_host = GF_host_doscalc.outputs.retrieved.get_abs_path('')
ef, dos, dos_interpol = interpolate_dos(dospath_host, return_original=True)
dos, dos_interpol = dos[0], dos_interpol[0]
```

(continues on next page)

(continued from previous page)

```
# read in impurity DOS
from numpy import loadtxt
impdos0 = loadtxt(kkrimp_doscalc.outputs.retrieved.get_abs_path('out_lmdos.interpol.
↪atom=01_spin1.dat'))
impdos1 = loadtxt(kkrimp_doscalc.outputs.retrieved.get_abs_path('out_lmdos.interpol.
↪atom=13_spin1.dat'))
# sum over spins:
impdos0[:,1:] = impdos0[:,1:]*2
impdos1[:,1:] = impdos1[:,1:]*2

# plot bulk and impurity DOS
from matplotlib.pyplot import figure, fill_between, plot, legend, title, axhline,␣
↪axvline, xlim, ylim, ylabel, xlabel, title, show
figure()
fill_between((dos_interpol[:,0]-ef)*13.6, dos_interpol[:,1]/13.6, color='lightgrey',␣
↪lw=0, label='bulk Cu')
plot((impdos0[:,0]-ef)*13.6, impdos0[:,1]/13.6, label='Au imp')
plot((impdos0[:,0]-ef)*13.6, impdos1[:,1]/13.6, label='1st Cu neighbor')
plot((impdos0[:,0]-ef)*13.6, (impdos1[:,1]-dos_interpol[:,1])/dos_interpol[:,1], '--',
↪ label='relative difference in 1st Cu neighbor')
legend()
title('DOS of Au impurity embedded into bulk Cu')
axhline(0, lw=1, color='grey')
axvline(0, lw=1, color='grey')
xlim(-8, 1)
ylim(-0.5,8.5)
xlabel('E-E_F (eV)')
ylabel('DOS (states/eV)')
show()
```

Which should look like this:



KKR calculation importer

Only functional in version below 1.0

Plugin `kkr.kkrimporter`

The calculation importer can be used to import a already finished KKR calculation to the aiida dbatabase. The KKRimporterCalculation takes the inputs

- `code`: KKR code installation on the computer from which the calculation is imported
- `computer`: computer on which the calulation has been performed
- `resources`: resources used in the calculation
- `remote_workdir`: remote abolute path on `computer` to the path where the calculation has been performed
- `input_file_names`: dictionary of input file names
- `output_file_names`, optional: dictionary of output file names

and mimicks a KKR calculation (i.e. stores KKR parameter set in node `parameters` and the extracted aiida StructureData node `structure` as inputs and creates `remote_folder`, `retrieved` and `output_parameters` output nodes). A KKRimporter calculation can then be used like a KKR claculation to continue calculations with correct provenance tracking in the database.

Note:

- At least `input_file` and `potential_file` need to be given in `input_file_names`.
- Works also if output was a Jij calculation, then `Jijatom.*` and `shells.dat` files are retreived as well.

Example on how to use the calculation importer:

```
# Load the KKRimporter class
from aiida.orm import CalculationFactory
KkrImporter = CalculationFactory('kkr.kkrimporter')

# Load the Code node representative of the one used to perform the calculations
from aiida.orm.code import Code
code = Code.get_from_string('KKRcode@my_mac')

# Get the Computer node representative of the one the calculations were run on
computer = code.get_remote_computer()

# Define the computation resources used for the calculations
resources = {'num_machines': 1, 'num_mpiprocs_per_machine': 1}

# Create calculation
calc1 = KkrImporter(computer=computer,
                    resources=resources,
                    remote_workdir='<absolute-remote-path-to-calculation>',
                    input_file_names={'input_file':'inputcard', 'potential_file':
↪'potential', 'shapefun_file':'shapefun'},
                    output_file_names={'out_potential_file':'potential'})

# Link the code that was used to run the calculations.
calc1.use_code(code)

# Get the computer's transport and create an instance.
from aiida.backends.utils import get_authinfo, get_automatic_user
authinfo = get_authinfo(computer=computer, aiidauser=get_automatic_user())
transport = authinfo.get_transport()
```

(continues on next page)

(continued from previous page)

```
# Open the transport for the duration of the immigrations, so it's not
# reopened for each one. This is best performed using the transport's
# context guard through the ``with`` statement.
with transport as open_transport:
    # Parse the calculations' input files to automatically generate and link the
    # calculations' input nodes.
    calc1.create_input_nodes(open_transport)

    # Store the calculations and their input nodes and tell the daeomon the output
    # is ready to be retrieved and parsed.
    calc1.prepare_for_retrieval_and_parsing(open_transport)
```

After the calculation has finished the following nodes should appear in the aiida database:

```
$ verdi calculation show <pk-to-imported-calculation>
-----------  ------------------------------------
type         KkrImporterCalculation
pk           22121
uuid         848c2185-8c82-44cd-ab67-213c20aaa414
label
description
ctime        2018-04-24 15:29:42.136154+00:00
mtime        2018-04-24 15:29:48.496421+00:00
computer     [1] my_mac
code         KKRcode
-----------  ------------------------------------
##### INPUTS:
Link label       PK  Type
------------  -----  -------------
parameters    22120  Dict
structure     22119  StructureData
##### OUTPUTS:
Link label            PK  Type
-----------------  -----  -------------
remote_folder      22122  RemoteData
retrieved          22123  FolderData
output_parameters  22124  Dict
##### LOGS:
There are 1 log messages for this calculation
Run 'verdi calculation logshow 22121' to see them
```

Example scripts

Here is a small collection of example scripts.

Scripts need to be updated for new version (>1.0)

Full example Voronoi-KKR-KKRimp

Compact script starting with structure setup, then voronoi calculation, followed by initial KKR claculation which is then continued for convergence. The converged calculation is then used to write out the host GF and a simple inmpurity calculation is performed.

Download: `this example script`

```
#!/usr/bin/env python

# connect to aiida db
from aiida import load_profile
load_profile()
# load essential aiida classes
from aiida.orm import Code
from aiida.orm import DataFactory
StructureData = DataFactory('structure')
Dict = DataFactory('parameter')

# load kkrparms class which is a useful tool to create the set of input parameters␣
↪for KKR-family of calculations
from aiida_kkr.tools.kkr_params import kkrparams

# load some python modules
from numpy import array

# helper function
def wait_for_it(calc, maxwait=300):
    from time import sleep
    N = 0
    print 'start waiting for calculation to finish'
    while not calc.has_finished() and N<(maxwait/2.):
        N += 1
        if N%5==0:
            print('.')
        sleep(2.)
    print('waiting done after {} seconds: {} {}'.format(N*2, calc.has_finished(),␣
↪calc.has_finished_ok()))


#####################################################
# initial structure
#####################################################

# create Copper bulk aiida Structure
alat = 3.61 # lattice constant in Angstroem
bravais = alat*array([[0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.5]]) # Bravais matrix␣
↪in Ang. units
Cu = StructureData(cell=bravais)
Cu.append_atom(position=[0,0,0], symbols='Cu')


#####################################################
# Voronoi step (preparation of starting potential)
#####################################################

# create empty set of KKR parameters (LMAX cutoff etc. ) for voronoi code
params = kkrparams(params_type='voronoi')

# and set at least the mandatory parameters
params.set_multiple_values(LMAX=2, NSPIN=1, RCLUSTZ=2.3)

# finally create an aiida Dict node and fill with the dictionary of parameters
ParaNode = Dict(dict=params.get_dict())
```

(continues on next page)

(continued from previous page)

```
# choose a valid installation of the voronoi code
### !!! adapt to your code name !!! ###
codename = 'voronoi@my_mac'
code = Code.get_from_string(codename)

# create new instance of a VoronoiCalculation
voro_calc = code.new_calc()

# and set resources that will be used (here serial job)
voro_calc.set_resources({'num_machines':1, 'tot_num_mpiprocs':1})

### !!! use queue name if necessary !!! ###
# voro_calc.set_queue_name('<quene_name>')

# then set structure and input parameter
voro_calc.use_structure(Cu)
voro_calc.use_parameters(ParaNode)

# store all nodes and submit the calculation
voro_calc.store_all()
voro_calc.submit()

wait_for_it(voro_calc)

# for future reference
voronoi_calc_folder = voro_calc.outputs.remote_folder
voro_params = voro_calc.inputs.parameters


##################################################
# KKR step (20 iterations simple mixing)
##################################################

# create new set of parameters for a KKR calculation and fill with values from␣
↪previous voronoin calculation
params = kkrparams(params_type='kkr', **voro_params.get_dict())

# and set the missing values
params.set_multiple_values(RMAX=7., GMAX=65.)

# choose 20 simple mixing iterations first to preconverge potential (here 5% simple␣
↪mixing)
params.set_multiple_values(NSTEPS=20, IMIX=0, STRMIX=0.05)

# create aiida Dict node from the KKR parameters
ParaNode = Dict(dict=params.get_dict())

# get KKR code and create new calculation instance
### !!! use your code name !!! ###
code = Code.get_from_string('KKRcode@my_mac')
kkr_calc = code.new_calc()

# set input Parameter, parent calulation (previous voronoi calculation), computer␣
↪resources
kkr_calc.use_parameters(ParaNode)
kkr_calc.use_parent_folder(voronoi_calc_folder)
kkr_calc.set_resources({'num_machines': 1, 'num_mpiprocs_per_machine':1})
```

(continues on next page)

(continued from previous page)

```
### !!! use queue name if necessary !!! ###
# kkr_calc.set_queue_name('<quene_name>')

# store nodes and submit calculation
kkr_calc.store_all()
kkr_calc.submit()

# wait for calculation to finish
wait_for_it(kkr_calc)


##################################################
# 2nd KKR step (continued from previous KKR calc)
##################################################

# create new KKR calculation instance to continue KKR ontop of a previous KKR␣
↪calclation
kkr_calc_continued = code.new_calc()

# reuse old KKR parameters and update scf settings (default is NSTEPS=1, IMIX=0)
params.set_multiple_values(NSTEPS=50, IMIX=5)
# and create aiida Dict node
ParaNode = Dict(dict=params.get_dict())

# then set input nodes for calculation
kkr_calc_continued.use_code(code)
kkr_calc_continued.use_parameters(ParaNode)
kkr_calc_parent_folder = kkr_calc.outputs.remote_folder # parent remote folder of␣
↪previous calculation
kkr_calc_continued.use_parent_folder(kkr_calc_parent_folder)
kkr_calc_continued.set_resources({'num_machines': 1, 'num_mpiprocs_per_machine':1})

### !!! use queue name if necessary !!! ###
# kkr_calc_continued.set_queue_name('<quene_name>')

# store input nodes and submit calculation
kkr_calc_continued.store_all()
kkr_calc_continued.submit()

# wait for calculation to finish
wait_for_it(kkr_calc_continued)


##################################################
# writeout host GF (using converged calculation)
##################################################

# take remote folder of converged calculation to reuse setting and write out Green␣
↪function and tmat of the crystalline host system
kkr_converged_parent_folder = kkr_calc_continued.outputs.remote_folder

# extreact kkr calculation from parent calculation folder
kkrcalc_converged = kkr_converged_parent_folder.get_inputs()[0]

# extract parameters from parent calculation and update RUNOPT for KKRFLEX option
kkr_params_dict = kkrcalc_converged.inputs.parameters.get_dict()
```

(continues on next page)

(continued from previous page)

```
kkr_params_dict['RUNOPT'] = ['KKRFLEX']

# create aiida Dict node with set parameters that are updated compared to converged␣
↪parent kkr calculation
ParaNode = Dict(dict=kkr_params_dict)

# create new KKR calculation
code = kkrcalc_converged.get_code() # take the same code as in the calculation before
GF_host_calc= code.new_calc()

# set resources, Parameter Node and parent calculation
resources = kkrcalc_converged.get_resources()
GF_host_calc.set_resources(resources)
GF_host_calc.use_parameters(ParaNode)
GF_host_calc.use_parent_folder(kkr_converged_parent_folder)

### !!! use queue name if necessary !!! ###
# GF_host_calc.set_queue_name('<quene_name>')

# prepare impurity_info node containing the information about the impurity cluster
imp_info = Dict(dict={'Rcut':1.01, 'ilayer_center':0, 'Zimp':[79.]})
# set impurity info node to calculation
GF_host_calc.use_impurity_info(imp_info)

# store input nodes and submit calculation
GF_host_calc.store_all()
GF_host_calc.submit()

# wait for calculation to finish
wait_for_it(GF_host_calc)


######################################################################
# KKRimp calculation (20 simple mixing iterations  for preconvergence)
######################################################################

# first create impurity start pot using auxiliary voronoi calculation

# creation of the auxiliary styructure:
# use an aiida workfunction to keep track of the provenance
from aiida.work import workfunction as wf
@wf
def change_struc_imp_aux_wf(struc, imp_info): # Note: works for single imp at center␣
↪only!
    from aiida.common.constants import elements as PeriodicTableElements
    _atomic_numbers = {data['symbol']: num for num, data in PeriodicTableElements.
↪iteritems()}

    new_struc = StructureData(cell=struc.cell)
    isite = 0
    for site in struc.sites:
        sname = site.kind_name
        kind = struc.get_kind(sname)
        pos = site.position
        zatom = _atomic_numbers[kind.get_symbols_string()]
        if isite == imp_info.get_dict().get('ilayer_center'):
            zatom = imp_info.get_dict().get('Zimp')[0]
```

(continues on next page)

(continued from previous page)

```
        symbol = PeriodicTableElements.get(zatom).get('symbol')
        new_struc.append_atom(position=pos, symbols=symbol)
        isite += 1

    return new_struc

new_struc = change_struc_imp_aux_wf(voro_calc.inputs.structure, imp_info)

# then Voronoi calculation for auxiliary structure
### !!! use your code name !!! ###
codename = 'voronoi@my_mac'
code = Code.get_from_string(codename)
voro_calc_aux = code.new_calc()
voro_calc_aux.set_resources({'num_machines':1, 'tot_num_mpiprocs':1})
voro_calc_aux.use_structure(new_struc)
voro_calc_aux.use_parameters(kkrcalc_converged.inputs.parameters)
voro_calc_aux.store_all()
voro_calc_aux.submit()
### !!! use queue name if necessary !!! ###
# voro_calc_aux.set_queue_name('<quene_name>')

# wait for calculation to finish
wait_for_it(voro_calc_aux)

# then create impurity startpot using auxiliary voronoi calc and converged host␣
↪potential

from aiida_kkr.tools.common_workfunctions import neworder_potential_wf

potname_converged = kkrcalc_converged._POTENTIAL
potname_imp = 'potential_imp'
neworder_pot1 = [int(i) for i in loadtxt(GF_host_calc.outputs.retrieved.get_abs_path(
↪'scoef'), skiprows=1)[:,3]-1]
potname_impvorostart = voro_calc_aux._OUT_POTENTIAL_voronoi
replacelist_pot2 = [[0,0]]

settings_dict = {'pot1': potname_converged,  'out_pot': potname_imp, 'neworder':␣
↪neworder_pot1,
                 'pot2': potname_impvorostart, 'replace_newpos': replacelist_pot2,
↪'label': 'startpot_KKRimp',
                 'description': 'starting potential for Au impurity in bulk Cu'}
settings = Dict(dict=settings_dict)

startpot_Au_imp_sfd = neworder_potential_wf(settings_node=settings,
                                            parent_calc_folder=kkrcalc_converged.out.
↪remote_folder,
                                            parent_calc_folder2=voro_calc_aux.out.
↪remote_folder)

# now create KKRimp calculation and run first (some simple mixing steps) calculation

# needed to link to host GF writeout calculation
GF_host_output_folder = GF_host_calc.out.remote_folder

# create new KKRimp calculation
from aiida_kkr.calculations.kkrimp import KkrimpCalculation
kkrimp_calc = KkrimpCalculation()
```

(continues on next page)

(continued from previous page)

```
### !!! use your code name !!! ###
kkrimp_code = Code.get_from_string('KKRimp@my_mac')

kkrimp_calc.use_code(kkrimp_code)
kkrimp_calc.use_host_Greenfunction_folder(GF_host_output_folder)
kkrimp_calc.use_impurity_potential(startpot_Au_imp_sfd)
kkrimp_calc.set_resources(resources)
kkrimp_calc.set_computer(kkrimp_code.get_computer())

# first set 20 simple mixing steps
kkrimp_params = kkrparams(params_type='kkrimp')
kkrimp_params.set_multiple_values(SCFSTEPS=20, IMIX=0, MIXFAC=0.05)
ParamsKKRimp = Dict(dict=kkrimp_params.get_dict())
kkrimp_calc.use_parameters(ParamsKKRimp)

# store and submit
kkrimp_calc.store_all()
kkrimp_calc.submit()

# wait for calculation to finish
wait_for_it(kkrimp_calc)


#####################################################
# continued KKRimp calculation until convergence
#####################################################

kkrimp_calc_converge = kkrimp_code.new_calc()
kkrimp_calc_converge.use_parent_calc_folder(kkrimp_calc.out.remote_folder)
kkrimp_calc_converge.set_resources(resources)
kkrimp_calc_converge.use_host_Greenfunction_folder(kkrimp_calc.inputs.GFhost_folder)

kkrimp_params = kkrparams(params_type='kkrimp', **kkrimp_calc.inputs.parameters.get_
↪dict())
kkrimp_params.set_multiple_values(SCFSTEPS=99, IMIX=5, MIXFAC=0.05)
ParamsKKRimp = Dict(dict=kkrimp_params.get_dict())
kkrimp_calc_converge.use_parameters(ParamsKKRimp)

### !!! use queue name if necessary !!! ###
# kkrimp_calc_converge.set_queue_name('<quene_name>')

# store and submit
kkrimp_calc_converge.store_all()
kkrimp_calc_converge.submit()

wait_for_it(kkrimp_calc_converge)
```

KKRimp DOS (starting from converged parent KKRimp calculation)

Script running host GF step for DOS contour first before running KKRimp step and plotting.

Download: `this example script`

```
#!/usr/bin/env python

# connect to aiida db
from aiida import load_profile
load_profile()
# load essential aiida classes
from aiida.orm import DataFactory, load_node
Dict = DataFactory('parameter')


# some settings:
#DOS contour (in Ry units), emax=EF+dE_emax:
emin, dE_emax, npt = -0.2, 0.1, 101
# kkrimp parent (converged imp pot, needs to tbe a KKRimp calculation node)
kkrimp_calc_converge = load_node(25025)

# derived quantities:
GF_host_calc = kkrimp_calc_converge.inputs.GFhost_folder.inputs.remote_folder
kkr_converged_parent_folder = GF_host_calc.inputs.parent_calc_folder

# helper function
def wait_for_it(calc, maxwait=300):
    from time import sleep
    N = 0
    print 'start waiting for calculation to finish'
    while not calc.has_finished() and N<(maxwait/2.):
        N += 1
        if N%5==0:
            print('.')
        sleep(2.)
    print('waiting done after {} seconds: {} {}'.format(N*2, calc.has_finished(),␣
↪calc.has_finished_ok()))

################################################################################
↪##########

# first host GF with DOS contour
from aiida_kkr.tools.kkr_params import kkrparams
params = kkrparams(**GF_host_calc.inputs.parameters.get_dict())
params.set_multiple_values(EMIN=emin, EMAX=GF_host_calc.res.fermi_energy+dE_emax,␣
↪NPOL=0, NPT1=0, NPT2=npt, NPT3=0)
ParaNode = Dict(dict=params.get_dict())

code = GF_host_calc.get_code() # take the same code as in the calculation before
GF_host_doscalc= code.new_calc()
resources = GF_host_calc.get_resources()
GF_host_doscalc.set_resources(resources)
GF_host_doscalc.use_parameters(ParaNode)
GF_host_doscalc.use_parent_folder(kkr_converged_parent_folder)
GF_host_doscalc.use_impurity_info(GF_host_calc.inputs.impurity_info)

# store and submit
GF_host_doscalc.store_all()
GF_host_doscalc.submit()

# wait for calculation to finish
print 'host GF calc for DOS contour'
```

(continues on next page)

(continued from previous page)

```
wait_for_it(GF_host_doscalc)

# then KKRimp step using the converged potential

kkrimp_doscalc = kkrimp_calc_converge.get_code().new_calc()
kkrimp_doscalc.use_host_Greenfunction_folder(GF_host_doscalc.out.remote_folder)
kkrimp_doscalc.use_parent_calc_folder(kkrimp_calc_converge.out.remote_folder)
kkrimp_doscalc.set_resources(kkrimp_calc_converge.get_resources())

# set to DOS settings
params = kkrparams(params_type='kkrimp', **kkrimp_calc_converge.inputs.parameters.get_
↪dict())
params.set_multiple_values(RUNFLAG=['lmdos'], SCFSTEPS=1)
ParaNode = Dict(dict=params.get_dict())

kkrimp_doscalc.use_parameters(ParaNode)

# store and submit calculation
kkrimp_doscalc.store_all()
kkrimp_doscalc.submit()

# wait for calculation to finish

print 'KKRimp calc DOS'
wait_for_it(kkrimp_doscalc)

# Finally plot the DOS:

# get interpolated DOS from GF_host_doscalc calculation:
from masci_tools.io.common_functions import interpolate_dos
dospath_host = GF_host_doscalc.out.retrieved.get_abs_path('')
ef, dos, dos_interpol = interpolate_dos(dospath_host, return_original=True)
dos, dos_interpol = dos[0], dos_interpol[0]

# read in impurity DOS
from numpy import loadtxt
impdos0 = loadtxt(kkrimp_doscalc.out.retrieved.get_abs_path('out_lmdos.interpol.
↪atom=01_spin1.dat'))
impdos1 = loadtxt(kkrimp_doscalc.out.retrieved.get_abs_path('out_lmdos.interpol.
↪atom=13_spin1.dat'))
# sum over spins:
impdos0[:,1:] = impdos0[:,1:]*2
impdos1[:,1:] = impdos1[:,1:]*2

# plot bulk and impurity DOS
from matplotlib.pyplot import figure, fill_between, plot, legend, title, axhline,␣
↪axvline, xlim, ylim, ylabel, xlabel, title, show
figure()
fill_between((dos_interpol[:,0]-ef)*13.6, dos_interpol[:,1]/13.6, color='lightgrey',␣
↪lw=0, label='bulk Cu')
plot((impdos0[:,0]-ef)*13.6, impdos0[:,1]/13.6, label='Au imp')
plot((impdos0[:,0]-ef)*13.6, impdos1[:,1]/13.6, label='1st Cu neighbor')
plot((impdos0[:,0]-ef)*13.6, (impdos1[:,1]-dos_interpol[:,1])/dos_interpol[:,1], '--',
↪ label='relative difference in 1st Cu neighbor')
legend()
title('DOS of Au impurity embedded into bulk Cu')
axhline(0, lw=1, color='grey')
```

(continues on next page)

(continued from previous page)

```
axvline(0, lw=1, color='grey')
xlim(-8, 1)
ylim(-0.5,8.5)
xlabel('E-E_F (eV)')
ylabel('DOS (states/eV)')
show()
```

KKR bandstructure

Script running a bandstructure calculation for which first from the structure node the kpoints of the high-symmetry lines are extracted and afterwards the bandstructure (i.e. `qdos`) calculation is started. Finally the results are plotted together with the DOS data (taken from KKRimp DOS preparation step).

Download: `this example script`

```
#!/usr/bin/env python

# connect to aiida db
from aiida import load_profile
load_profile()
# load essential aiida classes
from aiida.orm import Code, DataFactory, load_node
StructureData = DataFactory('structure')
Dict = DataFactory('parameter')

# helper function:
def wait_for_it(calc, maxwait=300):
    from time import sleep
    N = 0
    print 'start waiting for calculation to finish'
    while not calc.has_finished() and N<(maxwait/2.):
        N += 1
        if N%5==0:
            print('.')
        sleep(2.)
    print('waiting done after {} seconds: {} {}'.format(N*2, calc.has_finished(),␣
↪calc.has_finished_ok()))


# some settings (parent calculations):

# converged KKR calculation (taken form bulk Cu KKR example)
kkr_calc_converged = load_node(24951)
# previous DOS calculation started from converged KKR calc (taken from KKRimp DOS␣
↪example, i.e. GF host calculation with DOS contour)
host_dos_calc = load_node(25030)


# generate kpoints for bandstructure calculation

from aiida_kkr.calculations.voro import VoronoiCalculation
struc, voro_parent = VoronoiCalculation.find_parent_structure(kkr_calc_converged.out.
↪remote_folder)

from aiida.tools.data.array.kpoints import get_explicit_kpoints_path
```

(continues on next page)

(continued from previous page)

```
kpts = get_explicit_kpoints_path(struc).get('explicit_kpoints')


# run bandstructure calculation

# create bandstructure calculation reusing old settings (including same computer and␣
↪resources in this example)
kkrcode = kkr_calc_converged.get_code()
kkrcalc = kkrcode.new_calc()
kkrcalc.use_kpoints(kpts) # pass kpoints as input
kkrcalc.use_parent_folder(kkr_calc_converged.out.remote_folder)
kkrcalc.set_resources(kkr_calc_converged.get_resources())
# change parameters to qdos settings (E range and number of points)
from aiida_kkr.tools.kkr_params import kkrparams
qdos_params = kkrparams(**kkr_calc_converged.inputs.parameters.get_dict()) # reuse␣
↪old settings
# reuse the same emin/emax settings as in DOS run (extracted from input parameter␣
↪node)
qdos_params.set_multiple_values(EMIN=host_dos_calc.inputs.parameters.get_dict().get(
↪'EMIN'),
                                EMAX=host_dos_calc.inputs.parameters.get_dict().get(
↪'EMAX'),
                                NPT2=100)
kkrcalc.use_parameters(Dict(dict=qdos_params.get_dict()))

# store and submit calculation
kkrcalc.store_all()
kkrcalc.submit()

wait_for_it(kkrcalc, maxwait=600)


# plot results

# extract kpoint labels
klbl = kpts.labels
# fix overlapping labels (nicer plotting)
tmp = klbl[2]
tmp = (tmp[0], '\n'+tmp[1]+' ')
klbl[2] = tmp
tmp = klbl[3]
tmp = (tmp[0], '  '+tmp[1])
klbl[3] = tmp

#plotting of bandstructure and previously calculated DOS data

# load DOS data
from masci_tools.io.common_functions import interpolate_dos
dospath_host = host_dos_calc.out.retrieved.get_abs_path('')
ef, dos, dos_interpol = interpolate_dos(dospath_host, return_original=True)
dos, dos_interpol = dos[0], dos_interpol[0]

# load qdos file and reshape
from numpy import loadtxt, sum, log
qdos_file = kkrcalc.out.retrieved.get_abs_path('qdos.01.1.dat')
q = loadtxt(qdos_file)
nepts = len(set(q[:,0]))
```

(continues on next page)

(continued from previous page)

```
data = q[:,5:].reshape(nepts, len(q)/nepts, -1)
e = (q[::len(q)/nepts, 0]-ef)*13.6

# plot bandstructure
from matplotlib.pyplot import figure, pcolormesh, show, xticks, ylabel, axhline,␣
↪axvline, gca, title, plot, ylim, xlabel, suptitle
figure(figsize=((8, 4.8)))
pcolormesh(range(len(q)/nepts), e, log(sum(abs(data), axis=2)), lw=0)
xticks([i[0] for i in klbl], [i[1] for i in klbl])
ylabel('E-E_F (eV)')
axhline(0, color='lightgrey', lw=1)
title('band structure')

# plot DOS on right hand side of bandstructure plot
axBand = gca()
from mpl_toolkits.axes_grid1 import make_axes_locatable
divider = make_axes_locatable(axBand)
axDOS = divider.append_axes("right", 1.2, pad=0.1, sharey=axBand)

plot(dos_interpol[:,1]/13.6, (dos_interpol[:,0]-ef)*13.6)

ylim(e.min(), e.max())

axhline(0, color='grey', lw=1)
axvline(0, color='grey', lw=1)

axDOS.yaxis.set_tick_params(labelleft=False, labelright=True, right=True, left=False)
xlabel('states/eV')

title('DOS')
suptitle(struc.get_formula(), fontsize=16)

show()
```

Workflows

This page can contain a short introduction to the workflows provided by `aiida-kkr`.

Density of states

The density of states (DOS) workflow `kkr_dos_wc` automatically sets the right parameters in the input of a KKR calculation to perform a DOS calculation. The specifics of the DOS energy contour are set via the `wf_parameters` input node which contains default values if no user input is given.

Note: The default values of the `wf_parameters` input node can be extraced using `kkr_dos_wc.get_wf_defaults()`.

Inputs:

- `kkr` (*aiida.orm.Code*): KKRcode using the `kkr.kkr` plugin
- `remote_data` (*RemoteData*): The remote folder of the (converged) calculation whose output potential is used as input for the DOS run

- `wf_parameters` (*ParameterData*, optional): Some settings of the workflow behavior (e.g. number of energy points in DOS contour etc.)
- `options` (*ParameterData*, optional): Some settings for the computer you want to use (e.g. *queue_name*, *use_mpi*, *resources*, ...)
- `label` (*str*, optional): Label of the workflow
- `description` (*str*, optional): Longer description of the workflow

Returns nodes:

- `dos_data` (*XyData*): The DOS data on the DOS energy contour (i.e. at some finite temperature)
- `dos_data_interpol` (*XyData*): The interpolated DOS from the line parallel to the real axis down onto the real axis
- `results_wf` (*ParameterData*): The output node of the workflow containing some information on the DOS run

---

Note: The *x* and *y* arrays of the `dos_data` output nodes can easily be accessed using:

```
x = dos_data_node.get_x()
y = dos_data_node.get_y()
```

where the returned list is of the form `[label, numpy-array-of-data, unit]` and the *y*-array contains entries for total DOS, s-, p-, d-, ..., and non-spherical contributions to the DOS, e.g.:

```
[(u'interpolated dos tot', array([[...]]), u'states/eV'),
 (u'interpolated dos s', array([[...]]), u'states/eV'),
 (u'interpolated dos p', array([[...]]), u'states/eV'),
 (u'interpolated dos d', array([[...]]), u'states/eV'),
 (u'interpolated dos ns', array([[...]]), u'states/eV')]
```

Note that the output data are 2D arrays containing the atom resolved DOS, i.e. the DOS values for all atoms in the unit cell.

Example Usage

We start by getting an installation of the KKRcode:

```
from aiida.orm import Code
kkrcode = Code.get_from_string('KKRcode@my_mac')
```

Next load the remote folder node of the previous calculation (here the *converged calculation of the Cu bulk test case*) from which we want to start the following DOS calculation:

```
# import old KKR remote folder
from aiida.orm import load_node
kkr_remote_folder = load_node(22852).out.remote_folder
```

Then we set some settings of the workflow parameters (this step is optional):

```
# create workflow settings
from aiida.orm import DataFactory
ParameterData = DataFactory('parameter')
```

(continues on next page)

(continued from previous page)

```
workflow_settings = ParameterData(dict={'dos_params':{'emax': 1, 'tempr': 200, 'emin
↪': -1,
                                                       'kmesh': [20, 20, 20], 'nepts':␣
↪81}})
```

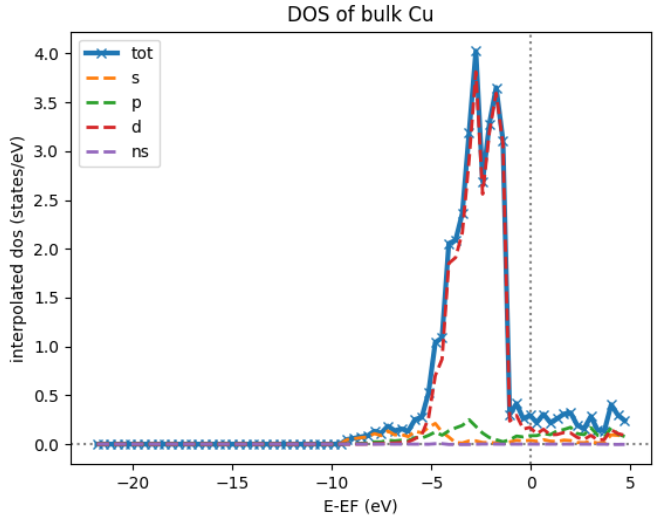
Finally we run the workflow:

```
from aiida_kkr.workflows.dos import kkr_dos_wc
from aiida.work import run
run(kkr_dos_wc, _label='test_doscal', _description='My test dos calculation.',
    kkr=kkrcode, remote_data=kkr_remote_folder, wf_parameters=workflow_settings)
```

The following script can be used to plot the total interpolated DOS (in the `dos_data_interpol` output node that can for example be access using `dos_data_interpol = <kkr_dos_wc-node>.out.dos_data_interpol` where `<kkr_dos_wc-node>` is the workflow node) of the calculation above:

```
def plot_dos(dos_data_node):
    x = dos_data_node.get_x()
    y_all = dos_data_node.get_y()

    from matplotlib.pylab import figure, xlabel, ylabel, axhline, axvline, plot,␣
↪legend, title

    figure()

    # loop over contributions (tot, s, p, d, ns)
    for y in y_all:
        if y==y_all[0]: # special line formatting for total DOS
            style = 'x-'
            lw = 3
        else:
            style = '--'
            lw = 2
        plot(x[1][0], y[1][0], style, lw=lw, ms=6, label=y[0].split('dos ')[1])

    # add axis labels etc
    xlabel(x[0]+' ({})'.format(x[-1]))
    ylabel(y[0].replace(' ns','')+' ({})'.format(y[-1]))
    axhline(0, color='grey', linestyle='dotted', zorder=-100)
    axvline(0, color='grey', linestyle='dotted', zorder=-100)
    legend(loc=2)
    title('DOS of bulk Cu')

plot_dos(dos_data_interpol)
```

which will produce the following plot:



Bandstructure

The bandstructure calculation, using workchain `kkr_bs_wc`, yields the band structure in terms of the Bloch spectral function. To run the bandstructure calculation all the requried parameters are taken from the parent (converved) KkrCalculation and user-defined `wf_parameters`.

Note: Use `kkr_bs_wc.get_wf_defaults()` to get the default values for the `wf_parameters` input.

Inputs:

- `wf_parameters` (Dict, optional): Workchain Specifications, contains `nepts` (int), `tempr` (float), `emin` (eV), `emax` (eV), `rclustz` (float, in units of the lattice constant). The energy range given by `emin` and `emax` are given relative to the Fermi level.
- `options` (Dict, optional): Computer Specifications, schedualer command, parallelization, walltime etc.
- `kpoints` (KpointsData, optional): k-point path used in the bandstructure calculation. If it is not given it is extructed from the structure. (Although it is important the k-points should come from the primitive structure, internally it will be consider in the next version.)
- `remote_data` (RemoteData, mendaory): Parent folder of a converged KkrCalculation.
- `kkr` (Code, mendaory): KKRhost code (i.e. using `kkr.kkr` plugin).
- `label` (Str, optional): label for the bandstructure WorkChainNode. Can also be found in the `result_wf` output Dict as `BS_wf_label` key.
- `description` (Str, optional) : description for the bandstructure WorkChainNode. Can be found in the `result_wf` output Dict as `BS_wf_description` key

Returns nodes:

- `BS_Data` (ArrayData): Consist of (BlochSpectralFunction, numpy array), (k_points, numpy array), (energy_points, numpy array), (special_kpoints, dict)
- `result_wf` (Dict): work_chain_specifications (such as ‘ *successful* ’, ‘ *list_of_errors* ’, ‘ *BS_params* ’ etc) node , *BS_data* (‘ *BlochSpectralFunction* ’,‘ *Kpts* ’,‘ *energy_points* ’, ‘ *k-labels* ’) node.

Access To Data:

To access into the data

```
BS_Data = <WC_NODE>.outputs.BS_Data
bsf = BS_Data.get_array('BlochSpectralFunction')
kpts = BS_Data.get_array('Kpts')
eng_pts = BS_Data.get_array('energy_points')
k_label= BS_Data.extras['k-labels']
```

The `bsf` array is a 2d-numpy array and contains the Bloch spectral function (k and energy resolved density) and `k_label` give the python dict archiving the high-symmetry points, `index:label`, in kpts.

Example Usage:

To start the Band Structure calculation the steps:

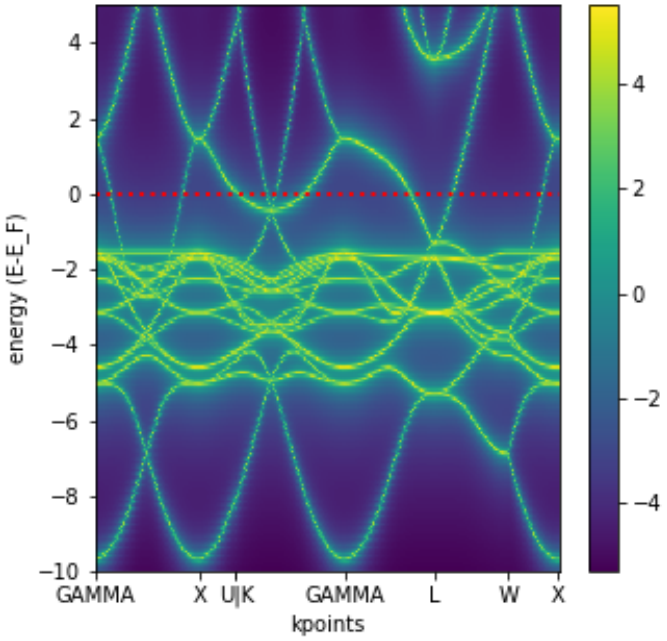
```
from aiida.orm import load_node, Str, Code, Dict

# setup the code and computer
kkrcode = Code.get_from_string('KKRcode@COMPUTERNAME')

# import the remote folder from the old converged kkr calculation
kkr_remote_folder = load_node(<KKR_CALC_JOB_NODE_ID>).outputs.remote_folder

# create workflow parameter settings
workflow_parameters = Dict(dict={'emax': 5, # in eV, relative to EF
                                 'tempr': 50.0, # in K
                                 'emin': -10, # in eV
                                 'rclustz' : 2.3, # alat units
                                 'nepts': 6})

# Computer configuration
metadata_option_1 = Dict(dict={
    'max_wallclock_seconds': 36000,
    'resources': {'tot_num_mpiprocs': 48, 'num_machines': 1},
    'custom_scheduler_commands':
    '#SBATCH --account=jara0191\n\nulimit -s unlimited; export OMP_STACKSIZE=2g',
    'withmpi': True})

label = Str('testing_the_kkr_bs_wc')

inputs = {'wf_parameters':workflow_parameters,'options':metadata_option_1,'remote_data
↪':kkr_remote_folder,'kkr':kkrcode,'label':label}

from aiida_kkr.workflows.bs import kkr_bs_wc
from aiida.engine import run
run(kkr_bs_wc, **inputs)
```

To plot :

To plot one or more kkr_bs_wc node.

```
from aiida import load_profile
load_profile()
NODE =  <singel or list of nodes>
from aiida_kkr.tools import plot_kkr
plot_kkr( NODE, strucplot=False, logscale=True, silent=True, noshow=True)
```

For bulk Cu this results in a plot like this:



Generate KKR start potential

Workflow: `kkr_startpot_wc`

Inputs:

- `structure` (*StructureData*):
- `voronoi` (*Code*):
- `kkr` (*Code*):
- `wf_parameters` (*ParameterData*, optional):
- `options` (*ParameterData*, optional): Some settings for the computer you want to use (e.g. *queue_name*, *use_mpi*, *resources*, ...)
- `calc_parameters` (*ParameterData*, optional):
- `label` (*str*, optional):
- `description` (*str*, optional):

Note: The default values of the `wf_parameters` input node can be extraced using `kkr_dos_wc.get_wf_defaults()` and it should contain the following entries:

General settings:

- `r_cls` (*float*):
- `natom_in_cls_min` (*int*):
- `fac_cls_increase` (*float*):
- `num_rerun` (*int*):

Computer settings:

- `walltime_sec` (*int*):
- `custom_scheduler_commands` (*str*):
- `use_mpi` (*bool*):
- `queue_name` (*str*):
- `resources` (*dict*): `{'num_machines': 1}`

Settings for DOS check of starting potential:

- `check_dos` (*bool*):
- `threshold_dos_zero` (*float*):
- `delta_e_min` (*float*):
- `delta_e_min_core_states` (*float*):
- **`dos_params` (*dict*): with the keys**
  - `emax` (*float*):
  - `tempr` (*float*):
  - `emin` (*float*):
  - `kmesh` ([*int*, *int*, *int*]):
  - `nepts` (*int*):

Output nodes:

- `last_doscal_dosdata` (*XyData*):
- `last_doscal_dosdata_interpol` (*XyData*):
- `last_doscal_results` (*ParameterData*):
- `last_params_voronoi` (*ParameterData*):
- `last_voronoi_remote` (*RemoteData*):
- `last_voronoi_results` (*ParameterData*):
- `results_vorostart_wc` (*ParameterData*):

Example Usage

First load KKRcode and Voronoi codes:

```
from aiida.orm import Code
kkrcode = Code.get_from_string('KKRcode@my_mac')
vorocode = Code.get_from_string('voronoi@my_mac')
```

Then choose some settings for the KKR specific parameters (LMAX cutoff etc.):

```
from aiida_kkr.tools.kkr_params import kkrparams
kkr_settings = kkrparams(NSPIN=1, LMAX=2)
```

Now we create a structure node for the system we want to calculate:

```
# create Copper bulk aiida Structure
from aiida.orm import DataFactory
StructureData = DataFactory('structure')
alat = 3.61 # lattice constant in Angstroem
bravais = alat*array([[0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.5]]) # Bravais matrix␣
↪in Ang. units
Cu = StructureData(cell=bravais)
Cu.append_atom(position=[0,0,0], symbols='Cu')
```

Finally we run the `kkr_startpot_wc` workflow (here using the defaults for the workflow settings):

```
from aiida_kkr.workflows.voro_start import kkr_startpot_wc
from aiida.work import run
ParameterData = DataFactory('parameter')
run(kkr_startpot_wc, structure=Cu, voronoi=vorocode, kkr=kkrcode, calc_
↪parameters=ParameterData(dict=kkr_settings.get_dict()))
```

KKR scf cycle

Workflow: `kkr_scf_wc`

Inputs:

```
{'strmix': 0.03, 'brymix': 0.05, 'init_pos': None, 'convergence_criterion': 1e-08,
 'custom_scheduler_commands': '', 'convergence_setting_coarse': {'npol': 7, 'tempr':␣
↪1000.0,
                                                                 'n1': 3, 'n2': 11,
↪'n3': 3,
                                                                 'kmesh': [10, 10,␣
↪10]},
 'mixreduce': 0.5, 'mag_init': False, 'retreive_dos_data_scf_run': False,
 'dos_params': {'emax': 0.6, 'tempr': 200, 'nepts': 81, 'kmesh': [40, 40, 40], 'emin
↪': -1},
 'hfield': 0.02, 'queue_name': '', 'threshold_aggressive_mixing': 0.008,
 'convergence_setting_fine': {'npol': 5, 'tempr': 600.0, 'n1': 7, 'n2': 29, 'n3': 7,
                              'kmesh': [30, 30, 30]},
 'use_mpi': False, 'nsteps': 50, 'resources': {'num_machines': 1}, 'delta_e_min': 1.0,
 'walltime_sec': 3600, 'check_dos': True, 'threshold_switch_high_accuracy': 0.001,
 'kkr_runmax': 5, 'threshold_dos_zero': 0.001}

_WorkChainSpecInputs({'_label': None, '_description': None, '_store_provenance': True,
                      'dynamic': None, 'calc_parameters': None, 'kkr': None, 'voronoi
↪': None,
                      'remote_data': None, 'wf_parameters': <ParameterData: uuid:␣
↪b132dfc4-3b7c-42e7-af27-4083802aff40 (unstored)>,
```

(continues on next page)

(continued from previous page)

```
                        'structure': None})
```

Outputs:

```
{'final_dosdata_interpol': <XyData: uuid: 0c14146d-90aa-4eb8-834d-74a706e500bb (pk:␣
↪22872)>,
 'last_InputParameters': <ParameterData: uuid: 28a277ad-8998-4728-8296-75fd3b0c4eb4␣
↪(pk: 22875)>,
 'last_RemoteData': <RemoteData: uuid: d24cdfc1-938a-4308-b273-e0aa8697c975 (pk:␣
↪22876)>,
 'last_calc_out': <ParameterData: uuid: 1c8fab2d-e596-4874-9516-c1387bf7db7c (pk:␣
↪22874)>,
 'output_kkr_scf_wc_ParameterResults': <ParameterData: uuid: 0f21ac18-e556-49f8-aa26-
↪55260d013fac (pk: 22878)>,
 'results_vorostart': <ParameterData: uuid: 93831550-8775-493a-907b-27a470b52dc8 (pk:␣
↪22877)>,
 'starting_dosdata_interpol': <XyData: uuid: 54fa57ad-f559-4837-ba1e-7db4ed67d5b0␣
↪(pk: 22873)>}
```

Example Usage

Case 1: Start from previous calculation

```
from aiida.orm import Code
kkrcode = Code.get_from_string('KKRcode@my_mac')
vorocode = Code.get_from_string('voronoi@my_mac')
```

```
from aiida_kkr.tools.kkr_params import kkrparams
kkr_settings = kkrparams(NSPIN=1, LMAX=2)
```

```
from aiida.orm import load_node
kkr_startpot = load_node(22586)
last_vorono_remote = kkr_startpot.get_outputs_dict().get('last_voronoi_remote')
```

```
from aiida_kkr.workflows.kkr_scf import kkr_scf_wc
from aiida.work import run
ParameterData = DataFactory('parameter')
run(kkr_scf_wc, kkr=kkrcode, calc_parameters=ParameterData(dict=kkr_settings.get_
↪dict()), remote_data=last_vorono_remote)
```

Case 2: Start from structure and run voronoi calculation first

```
# create Copper bulk aiida Structure
fro numpy import array
from aiida.orm import DataFactory
StructureData = DataFactory('structure')
alat = 3.61 # lattice constant in Angstroem
bravais = alat*array([[0.5, 0.5, 0], [0.5, 0, 0.5], [0, 0.5, 0.5]]) # Bravais matrix␣
↪in Ang. units
Cu = StructureData(cell=bravais)
Cu.append_atom(position=[0,0,0], symbols='Cu')
```

```
run(kkr_scf_wc, structure=Cu, kkr=kkrcode, voronoi=vorocode, calc_
↪parameters=ParameterData(dict=kkr_settings.get_dict()))
```

KKR flex (GF calculation)

The Green's function writeout workflow performs a KKR calculation with runoption `KKRFLEX` to write out the `kkr_flexfiles`. Those are needed for a `kkrimp` calculation.

Inputs:

- `kkr` (*aiida.orm.Code*): KKRcode using the `kkr.kkr` plugin
- `remote_data` (*RemoteData*): The remote folder of the (converged) kkr calculation
- `imp_info` (*ParameterData*): ParameterData node containing the information of the desired impurities (needed to write out the `kkr_flexfiles` and the `scoef` file)
- `options` (*ParameterData*, optional): Some settings for the computer (e.g. computer settings)
- `wf_parameters` (*ParameterData*, optional): Some settings for the workflow behaviour
- `label` (*str*, optional): Label of the workflow
- `description` (*str*, optional): Longer description of the workflow

Returns nodes:

- `workflow_info` (*ParameterData*): Node containing general information about the workflow (e.g. errors, computer information, ...)
- `GF_host_remote` (*RemoteData*): RemoteFolder with all of the `kkrflexfiles` and further output of the workflow

Example Usage

We start by getting an installation of the KKRcode:

```
from aiida.orm import Code
kkrcode = Code.get_from_string('KKRcode@my_mac')
```

Next load the remote folder node of the previous calculation (here the *converged calculation of the Cu bulk test case*) from which we want to start the following KKRFLEX calculation:

```
# import old KKR remote folder
from aiida.orm import load_node
kkr_remote_folder = load_node(<pid of converged calc>).out.remote_folder
```

Afterwards, the information regarding the impurity has to be given (in this example, we use a Au impurity with a cutoff radius of 2 alat which is placed in the first labelled lattice point of the unit cell). Further keywords for the `impurity_info` node can be found in the respective part of the documentation:

```
# set up impurity info node
imps = ParameterData(dict={'ilayer_center':0, 'Rcut':2, 'Zimp':[79.]})
```

Then we set some settings of the options parameters (this step is optional):

```
# create workflow settings
from aiida.orm import DataFactory
ParameterData = DataFactory('parameter')
options = ParameterData(dict={'use_mpi':'false', 'queue_name':'viti_node', 'walltime_
↪sec' : 60*60*2,
                                        'resources':{'num_machines':1, 'num_mpiprocs_
↪per_machine':1}})
```

Finally we run the workflow:

```
from aiida_kkr.workflows.gf_writeout import kkr_flex_wc
from aiida.work import run
run(kkr_flex_wc, label='test_gf_writeout', description='My test KKRflex calculation.',
    kkr=kkrcode, remote_data=kkr_remote_folder, options=options, wf_parameters=wf_
↪params)
```

KKR impurity self consistency

This workflow performs a KKRimp self consistency calculation starting from a given host-impurity startpotential and converges it.

Note: This workflow does only work for a non-magnetic calculation without spin-orbit-coupling. Those two features will be added at a later stage. This is also just a sub workflow, meaning that it only converges an already given host-impurity potential. The whole kkrimp workflow starting from scratch will also be added at a later stage.

Inputs:

- `kkrimp` (*aiida.orm.Code*): KKRimpcode using the `kkr.kkrimp` plugin
- `host_imp_startpot` (*SinglefileData*, optional): File containing the host impurity potential (potential file with the whole cluster with all host and impurity potentials)
- `remote_data` (*RemoteData*, optional): Output from a KKRflex calculation (can be extracted from the output of the GF writeout workflow)
- `kkrimp_remote` (*RemoteData*, optional): RemoteData output from previous kkrimp calculation (if given, `host_imp_startpot` is not needed as input)
- `impurity_info` (*ParameterData*, optional): Node containing information about the impurity cluster (has to be chosen consistently with `imp_info` from GF writeout step)
- `options` (*ParameterData*, optional): Some general settings for the workflow (e.g. computer settings, queue, ...)
- `wf_parameters` (*ParameterData*, optional) : Settings for the behavior of the workflow (e.g. convergence settings, physical properties, ...)
- `label` (*str*, optional): Label of the workflow
- `description` (*str*, optional): Longer description of the workflow

Returns nodes:

- `workflow_info` (*ParameterData*): Node containing general information about the workflow (e.g. errors, computer information, ...)
- `host_imp_pot` (*SinglefileData*): Converged host impurity potential that can be used for further calculations (DOS calc, new input for different KKRimp calculation)

Example Usage

We start by getting an installation of the KKRimpcode:

```
from aiida.orm import Code
kkrimpcode = Code.get_from_string('KKRimpcode@my_mac')
```

Next, either load the remote folder node of the previous calculation (here the KKRflex calculation that writes out the GF and KKRflexfiles) or the output node of the gf_writeout workflow from which we want to start the following KKRimp calculation:

```
# import old KKRFLEX remote folder
from aiida.orm import load_node
GF_host_output_folder = load_node(<pid of KKRFLEX calc>).out.remote_folder # 1st␣
↪possibility
# GF_host_output_folder = load_node(<pid of gf_writeout wf output node>) # 2nd␣
↪possibility: take ``GF_host_remote`` output node from gf_writeout workflow
```

Now, load a converged calculation of the host system (here Cu bulk) as well as an auxiliary voronoi calculation (here Au) for the desired impurity:

```
# load converged KKRcalc
kkrcalc_converged = load_node(<pid of converged KKRcalc (Cu bulk)>)
# load auxiliary voronoi calculation
voro_calc_aux = load_node(<pid of voronoi calculation for the impurity (Au)>)
```

Using those, one can obtain the needed host-impurity potential that is needed as input for the workflow. Therefore, we use the `neworder_potential_wf` workfunction which is able to generate the startpot:

```
## load the neccessary function
from aiida_kkr.tools.common_workfunctions import neworder_potential_wf
import numpy as np

# extract the name of the converged host potential
potname_converged = kkrcalc_converged._POTENTIAL
# set the name for the potential of the desired impurity (here Au)
potname_imp = 'potential_imp'

neworder_pot1 = [int(i) for i in np.loadtxt(GF_host_calc.out.retrieved.get_abs_path(
↪'scoef'), skiprows=1)[:,3]-1]
potname_impvorostart = voro_calc_aux._OUT_POTENTIAL_voronoi
replacelist_pot2 = [[0,0]]

# set up settings node to use as argument for the neworder_potential function
settings_dict = {'pot1': potname_converged,  'out_pot': potname_imp, 'neworder':␣
↪neworder_pot1,
                 'pot2': potname_impvorostart, 'replace_newpos': replacelist_pot2,
↪'label': 'startpot_KKRimp',
                 'description': 'starting potential for Au impurity in bulk Cu'}
settings = ParameterData(dict=settings_dict)

# finally create the host-impurity potential (here ``startpot_Au_imp_sfd``) using the␣
↪settings node as well as
the previously loaded converged KKR calculation and auxiliary voronoi calculation:
startpot_Au_imp_sfd = neworder_potential_wf(settings_node=settings,
                                            parent_calc_folder=kkrcalc_converged.out.
↪remote_folder,
```

(continues on next page)

(continued from previous page)

```
                                                    parent_calc_folder2=voro_calc_aux.out.
↪remote_folder)
```

Note: Further information how the neworder potential function works can be found in the respective part of this documentation.

Afterwards, the information regarding the impurity has to be given (in this example, we use a Au impurity with a cutoff radius of 2 alat which is placed in the first labelled lattice point of the unit cell). Further keywords for the `impurity_info` node can be found in the respective part of the documentation:

```
# set up impurity info node
imps = ParameterData(dict={'ilayer_center':0, 'Rcut':2, 'Zimp':[79.]})
```

Then, we set some settings of the options parameters on the one hand and specific wf_parameters regarding the convergence etc.:

```
options = ParameterData(dict={'use_mpi':'false', 'queue_name':'viti_node', 'walltime_
↪sec' : 60*60*2,
                              'resources':{'num_machines':1, 'num_mpiprocs_per_machine
↪':20}})
kkrimp_params = ParameterData(dict={'nsteps': 50, 'convergence_criterion': 1*10**-8,
↪'strmix': 0.1,
                                    'threshold_aggressive_mixing': 3*10**-2,
↪'aggressive_mix': 3,
                                     'aggrmix': 0.1, 'kkr_runmax': 5})
```

Finally we run the workflow:

```
from aiida_kkr.workflows.kkr_imp_sub import kkr_imp_sub_wc
from aiida.work import run
run(kkr_imp_sub_wc, label='kkr_imp_sub test (CuAu)', description='test of the kkr_imp_
↪sub workflow for Cu, Au system',
    kkrimp=kkrimpcode, options=options, host_imp_startpot=startpot_Au_imp_sfd,
    remote_data=GF_host_output_folder, wf_parameters=kkrimp_params)
```

KKR impurity workflow

This workflow performs a KKR impurity calculation starting from an `impurity_info` node as well as either from a coverged calculation remote for the host system (1) or from a GF writeout remote (2). In the two cases the following is done:

- (1): First, the host system will be converged using the `kkr_scf` workflow. Then, the GF will be calculated using the `gf_writeout` workflow before calculating the auxiliary startpotential of the impurity. Now, the total impurity-host startpotential will be generated and then converged using the `kkr_imp_sub` workflow.
- (2): In this case the two first steps from above will be skipped and the workflow starts by calculating the auxiliary startpotential.

Note: This workflow is different from the `kkr_imp_sub` workflow that only converges a given impurity host potential. Here, the whole process of a KKR impurity calculation is done automatically.

Inputs:

- `kkrimp` (*aiida.orm.Code*): KKRimpcode using the `kkr.kkrimp` plugin
- `voronoi` (*aiida.orm.Code*): Voronoi code using the `kkr.voro` plugin
- `kkr` (*aiida.orm.Code*): KKRhost code using the `kkr.kkr` plugin
- `impurity_info` (*ParameterData*): Node containing information about the impurity cluster
- `remote_data_host` (*RemoteData*, optional): RemoteData of a converged host calculation if you want to start the workflow from scratch
- `remote_data_gf` (*RemoteData*, optional): RemoteData of a GF writeout step (if you want to skip the convergence of the host and the GF writeout step)
- `options` (*ParameterData*, optional): Some general settings for the workflow (e.g. computer settings, queue, ...)
- `wf_parameters` (*ParameterData*, optional) : Settings for the behavior of the workflow (e.g. convergence settings, physical properties, ...)
- `voro_aux_parameters` (*ParameterData*, optional): Settings for the usage of the `kkr_startpot` sub workflow needed for the auxiliary voronoi potentials
- `label` (*str*, optional): Label of the workflow
- `description` (*str*, optional): Longer description of the workflow

Returns nodes:

- `workflow_info` (*ParameterData*): Node containing general information about the workflow
- `last_calc_info` (*ParameterData*): Node containing information about the last used calculation of the workflow
- `last_calc_output_parameters` (*ParameterData*): Node with all of the output parameters from the last calculation of the workflow

Example Usage

We start by getting an installation of the codes:

```
from aiida.orm import Code
kkrimpcode = Code.get_from_string('KKRimpcode@my_mac')
kkrcode = Code.get_from_string('KKRcode@my_mac')
vorocode = Code.get_from_string('vorocode@my_mac')
```

Then, set up an appropriate `impurity_info` node for your calculation:

```
# set up impurity info node
imps = ParameterData(dict={'ilayer_center':0, 'Rcut':2, 'Zimp':[79.]})
```

Next, load either a `gf_writeout_remote` or a `converged_host_remote`:

```
from aiida.orm import load_node
gf_writeout_remote = load_node(<pid or uuid>)
converged_host_remote = load_node(<pid or uuid>)
```

Set up some more input parameter nodes for your workflow:

```
# node for general workflow options
options = ParameterData(dict={'use_mpi': False, 'walltime_sec' : 60*60*2,
                              'resources':{'num_machines':1, 'num_mpiprocs_per_machine
↪':1}})
# node for convergence behaviour of the workflow
kkrimp_params = ParameterData(dict={'nsteps': 99, 'convergence_criterion': 1*10**-8,
↪'strmix': 0.02,
                                    'threshold_aggressive_mixing': 8*10**-2,
↪'aggressive_mix': 3,
                                    'aggrmix': 0.04, 'kkr_runmax': 5, 'calc_orbmom':␣
↪False, 'spinorbit': False,
                                    'newsol': False, 'mag_init': False, 'hfield': [0.
↪05, 10],
                                    'non_spherical': 1, 'nspin': 2})
# node for parameters needed for the auxiliary voronoi workflow
voro_aux_params = ParameterData(dict={'num_rerun' : 4, 'fac_cls_increase' : 1.5,
↪'check_dos': False,
                                      'lmax': 3, 'gmax': 65., 'rmax': 7., 'rclustz':␣
↪2.})
```

Finally, we run the workflow (for the two cases depicted above):

```
from aiida_kkr.workflows.kkr_scf import kkr_scf_wc
from aiida_kkr.workflows.voro_start import kkr_startpot_wc
from aiida_kkr.workflows.kkr_imp_sub import kkr_imp_sub_wc
from aiida_kkr.workflows.gf_writeout import kkr_flex_wc
from aiida_kkr.workflows.kkr_imp import kkr_imp_wc
from aiida.work.launch import run, submit

# don't forget to set a label and description for your workflow

# case (1)
wf_run = submit(kkr_imp_wc, label=label, description=description, voronoi=vorocode,␣
↪kkrimp=kkrimpcode,
                kkr=kkrcode, options=options, impurity_info=imps, wf_
↪parameters=kkrimp_params,
                voro_aux_parameters=voro_aux_params, remote_data_gf=gf_writeout_
↪remote)

# case (2)
wf_run = submit(kkr_imp_wc, label=label, description=description, voronoi=vorocode,␣
↪kkrimp=kkrimpcode,
                kkr=kkrcode, options=options, impurity_info=imps, wf_
↪parameters=kkrimp_params,
                voro_aux_parameters=voro_aux_params, remote_data_host=converged_host_
↪remote)
```

KKR impurity density of states

This workflow calculates the density of states for a given host impurity input potential.

Inputs:

- `kkrimp` (*aiida.orm.Code*): KKRimpcode using the `kkr.kkrimp` plugin
- `kkr` (*aiida.orm.Code*): KKRhost code using the `kkr.kkr` plugin
- `host_imp_pot` (*SinglefileData*): converged host impurity potential from impurity workflow

- `options` (*ParameterData*, optional): Some general settings for the workflow (e.g. computer settings, queue, ...)
- `wf_parameters` (*ParameterData*, optional) : Settings for the behavior of the workflow (e.g. convergence settings, physical properties, ...)
- `label` (*str*, optional): Label of the workflow
- `description` (*str*, optional): Longer description of the workflow

Returns nodes:

- `workflow_info` (*ParameterData*): Node containing general information about the workflow
- `last_calc_info` (*ParameterData*): Node containing information about the last used calculation of the workflow
- `last_calc_output_parameters` (*ParameterData*): Node with all of the output parameters from the last calculation of the workflow

Example Usage

We start by getting an installation of the codes:

```
from aiida.orm import Code
kkrimpcode = Code.get_from_string('KKRimpcode@my_mac')
vorocode = Code.get_from_string('vorocode@my_mac')
```

Next, load the converged host impurity potential:

```
from aiida.orm import load_node
startpot = load_node(<pid or uuid of SinglefileData>)
```

Set up some more input parameter nodes for your workflow:

```
# node for general workflow options
options = ParameterData(dict={'use_mpi': False, 'walltime_sec' : 60*60*2,
                              'resources':{'num_machines':1, 'num_mpiprocs_per_machine
↪':1}})
# node for convergence behaviour of the workflow
wf_params = ParameterData(dict={'ef_shift': 0. ,
                                'dos_params': {'nepts': 61,
                                               'tempr': 200,
                                               'emin': -1,
                                               'emax': 1,
                                               'kmesh': [30, 30, 30]},
                                'non_spherical': 1,
                                'born_iter': 2,
                                'init_pos' : None,
                                'newsol' : False})
```

Finally, we run the workflow (for the two cases depicted above):

```
from aiida_kkr.workflows.kkr_imp_dos import kkr_imp_dos_wc
from aiida.work.launch import run, submit

# don't forget to set a label and description for your workflow
```

(continues on next page)

(continued from previous page)

```
wf_run = submit(kkr_imp_dos_wc, label=label, description=description,␣
↪kkrimp=kkrimpcode,
                kkrcode=kkrcode, options=options, wf_parameters=wf_params)
```

Equation of states

Workflow: `aiida_kkr.workflows.eos`

Warning: Not implemented yet!

Check KKR parameter convergence

Workflow: `aiida_kkr.workflows.check_para_convergence`

Warning: Not implemented yet!

Idea is to run checks after convergence for the following parameters:

- RMAX
- GMAX
- cluster radius
- energy contour
- kmesh

Find magnetic ground state

Workflow: `aiida_kkr.workflows.check_magnetic_state`

Warning: Not implemented yet!

The idea is to run a Jij calculation to estimate if the ferromagnetic state is the ground state or not. Then the unit cell could be doubled to compute the antiferromagnetic state. In case of noncollinear magnetism the full Jij tensor should be analyzed.

Workfunctions

Here the workfunctions provided by the aiida-kkr plugin are presented. The workfunctions are small tools useful for small tasks performed on aiida nodes that keep the provenance in the database.

update_params_wf

The workfunktion `aiida_kkr.tools.common_workfunctions.update_params_wf` takes as an input a *ParameterData* node (`parameternode`) containing a KKR parameter set (i.e. created using the `kkrparams` class) and updates the parameter node with new values given in the dictionary of the second *ParameterData* input node (`updatenode`).

Input nodes:

- `parameternode` (ParameterData): aiida node of a KKR parameter set
- `updatenode` (ParameterData): aiida node containing parameter names with new values

Output node:

- `updated_parameter_node` (ParameterData): new parameter node with updated values

---

Note: If the `updatenode` contains the keys `nodename` and/or `nodedesc` then the label and/or description of the output node will be set accordingly.

Example Usage:

```
# initial KKR parameter node
input_node = ParameterData(dict=kkrparams(LMAX=3, EMIN=0))
input_node.store()
# update some values (e.g. change EMIN)
updated_params = ParameterData(dict={'nodename': 'my_changed_name', 'nodedesc': 'My␣
↪description text', 'EMIN': -1, 'RMAX': 10.})
new_params_node = update_params_wf(input_node, updated_params)
```

neworder_potential_wf

The workfunction `aiida_kkr.tools.common_workfunctions.neworder_potential_wf` creates a *SingleFileData* node that contains the new potential based in a potential file in the *RemoteFolder* input node (`settings_node`) which is braught to a new order according to the workfunction settings in the *ParameterData* input node (`parent_calc_folder`).

Input nodes:

- `settings_node` (ParameterData): Settings like filenames and neworder-list
- `parent_calc_folder` (RemoteData): folder where initial potential file is found
- `parent_calc_folder2` (RemoteData, optional): folder where second potential is found

Output node:

- `potential_file` (SingleFileData): output potential in new order

Note:

The settings_dict should contain the following keys:

- `pot1`, mandatory: <*filename_input_potential*>
- `out_pot`, mandatory: <*filename_output_potential*>
- `neworder`, mandatory: *[list of intended order in output potential]*

- `pot2`, mandatory if `parent_calc_folder2` is given as input node: *<filename_second_input_file>*
- `replace_newpos`, mandatory if `parent_calc_folder2` is given as input node: *[[position in neworder list which is replace with potential from pot2, position in pot2 that is chosen for replacement]]*
- `label`, optional: *label_for_output_node*
- `description`, optional: *longer_description_for_output_node*

prepare_VCA_structure_wf

Warning: Not implemented yet!

prepare_2Dcalc_wf

Warning: Not implemented yet!

Tools

Here the tools provided by `aiida-kkr` are described.

Plotting tools

Visualize typical nodes using `plot_kkr` from `aiida_kkr.tools.plot_kkr`. The `plot_kkr` function takes a node reference (can be a pk, uuid or the node itself or a list of these) and creates common plots for a quick visualization of the results obtained with the `aiida-kkr` plugin.

Usage example:

```
from aiida_kkr.tools.plot_kkr import plot_kkr
# use pk:
plot_kkr(999999)
# use uuid:
plot_kkr('xxxxx-xxxxx')
# used actual aiida node:
from aiida.orm import load_node
plot_kkr(load_node(999999))
# give list of nodes which goups plots together
plot_kkr([999999, 999998, 'xxxx-xxxxx', load_node(999999)])
```

The behavior of `plot_kkr` can be controled using keyword arguments:

```
plot_kkr(99999, strucplot=False) # do not call ase`s view function to visualize␣
↪structure
plot_kkr(99999, silent=True) # plots only (no printout of inputs/outputs to node)
```

List of `plot_kkr` specific keyword arguments:

- `silent` (*bool*, default: `False`): print information about input node including inputs and outputs
- `strucplot` (*bool*, default: `True`): plot structure using ase's `view` function

- `interpol` (*bool*, default: `True`): use interpolated data for DOS plots
- `all_atoms` (*bool*, default: `False`): plot all atoms in DOS plots (default: plot total DOS only)
- `l_channels` (*bool*, default: `True`): plot l-channels in addition to total DOS
- `logscale` (*bool*, default: `True`): plot rms and charge neutrality curves on a log-scale

Other keyword arguments are passed onto plotting functions, e.g. to modify line properties etc. (see matplotlib documentation for a reference of possible keywords to modify line properties):

```
plot_kkr(99999, marker='o', color='r') # red lines with 'o' markers
```

Examples

Plot structure node

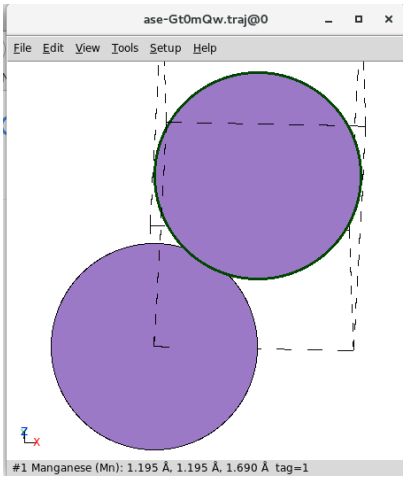


Fig. 1: Visualize a structure node (also happens as sub-parts of workflows that have a structure as input if `strucplot` is not set to `False`). Shown is a screenshot of the ouput produced by ase's `view`.

Plot output of a KKR calculation

Plot output of `kkr_dos_wc` workflow

Plot output of `kkr_startpot_wc` workflow

Plot output of `kkr_scf_wc` workflow

Plot output of `kkr_eos_wc` workflow

Plot multiple KKR calculations at once in the same plot

```
plot_kkr([34157,31962, 31974], silet=True, strucplot=False, logscale=False)
```

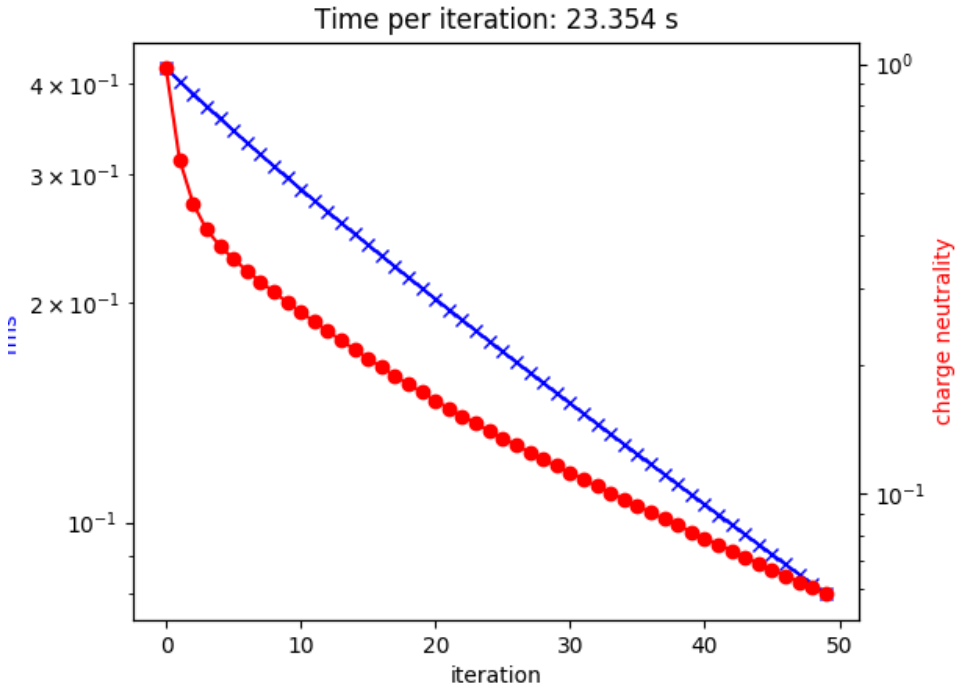


Fig. 2: Visualize the output of a `KkrCalculation`.

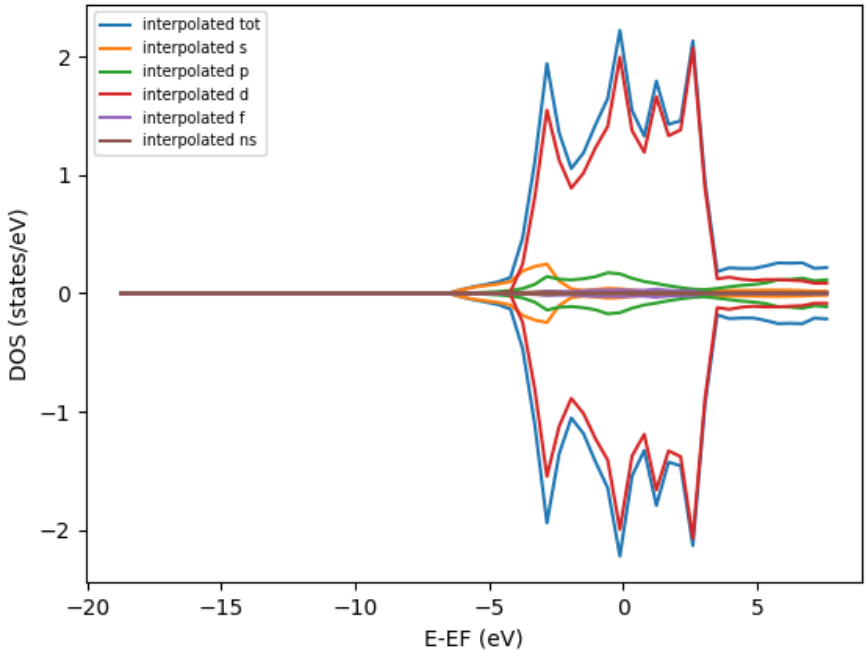


Fig. 3: Visualize the output of a `kkr_dos_wc` workflow.

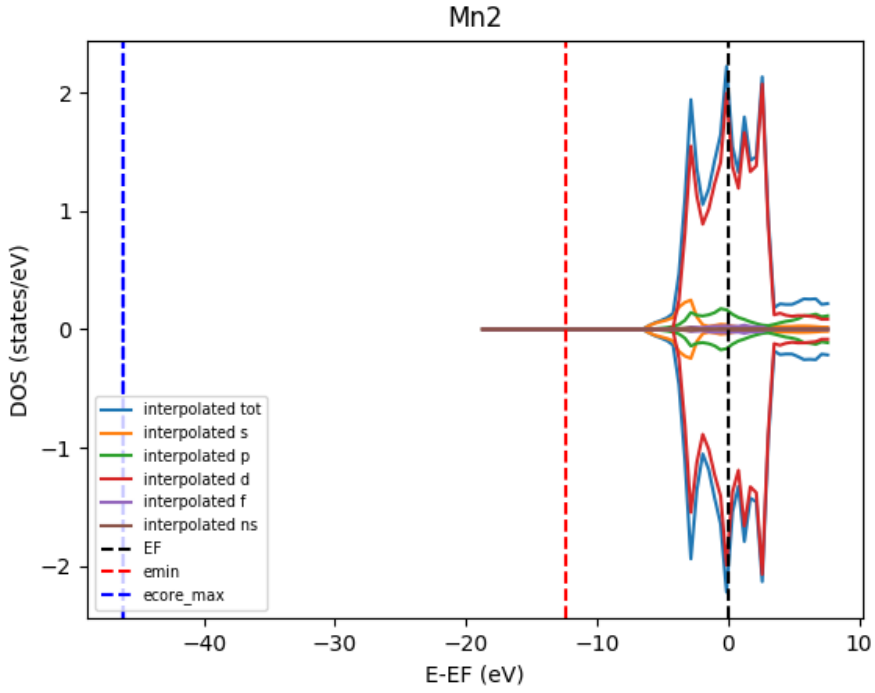


Fig. 4: Visualize the output of a `kkr_startpot_wc` workflow. The starting DOS is shown and the vertical lines indicate the position of the highest core states, the start of the ernergy contour and the Fermi level.

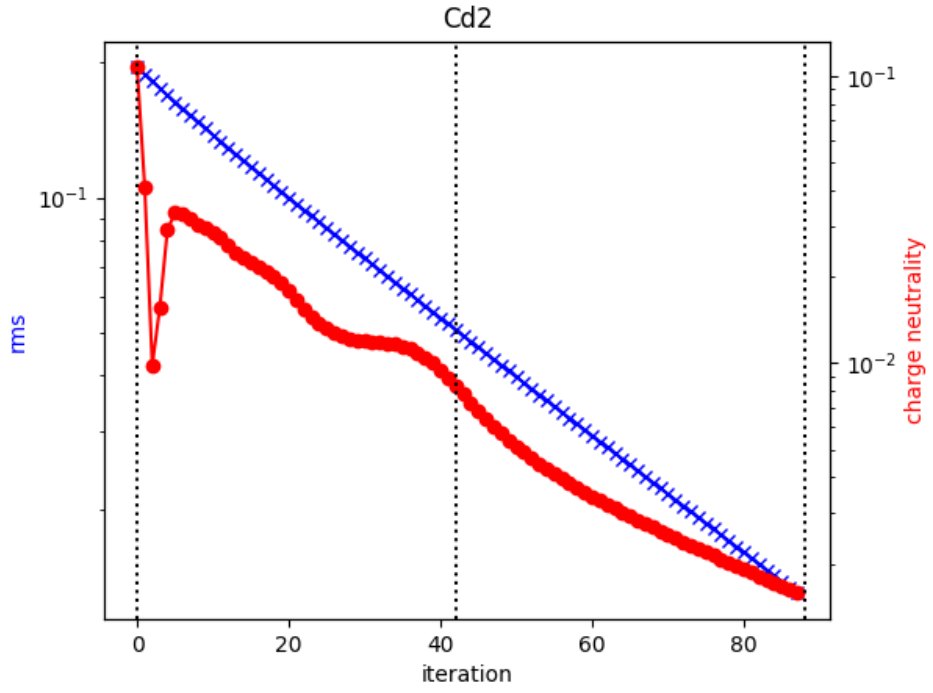


Fig. 5: Visualize the output of an unfinished `kkr_scf_wc` workflow. The vertical lines indicate where individual calculations have started and ended.

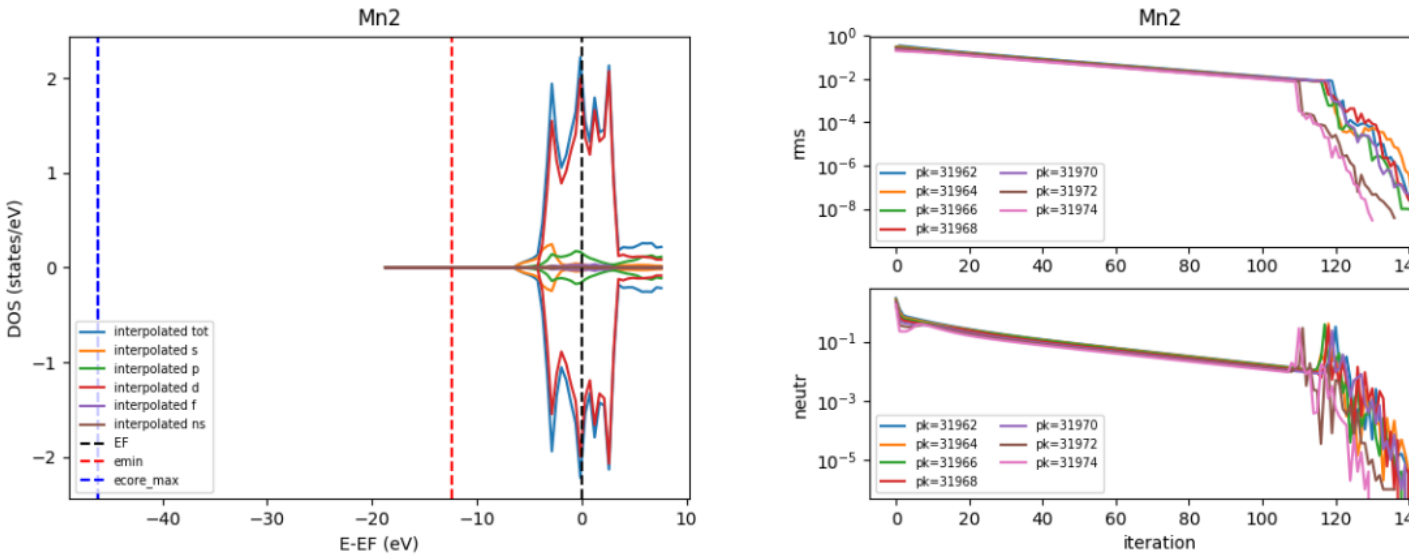


Fig. 6: Visualize the output of a `kkr_eos_wc` workflow.

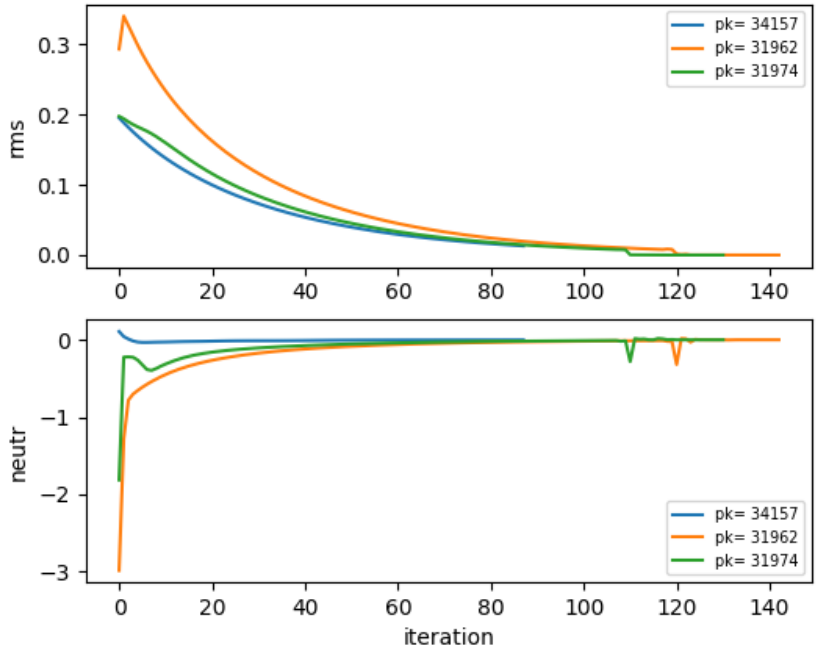


Fig. 7: Visualize the output of multiple `kkr_scf_wc` workflows without plotting structure.

1.1.2 Modules provided with aiida-kkr (API reference)

1.1.2.1 Modules provided with aiida-kkr (API reference)

Calculations

Voronoi

Input plug-in for a voronoi calculation.

class aiida_kkr.calculations.voro.**VoronoiCalculation**(**args*, ***kwargs*)

AiiDA calculation plugin for a voronoi calculation (creation of starting potential and shapefun).

_check_valid_parent(*parent_calc_folder*)

Check that calc is a valid parent for a FleurCalculation. It can be a VoronoiCalculation, KKRCalculation

classmethod _get_parent(*input_folder*)

get the parent folder of the calculation. If not parent was found return input folder

classmethod _get_remote(*parent_folder*)

get remote_folder from input if parent_folder is not already a remote folder

classmethod _get_struc(*parent_calc*)

Get structure from a parent_folder (result of a calculation, typically a remote folder)

classmethod _has_struc(*parent_folder*)

Check if parent_folder has structure information in its input

_is_KkrCalc(*calc*)

check if calc contains the file out_potential

classmethod define(*spec*)

define internals and inputs / outputs of calculation

classmethod find_parent_structure(*parent_folder*)

Find the Structure node recuresively in chain of parent calculations (structure node is input to voronoi calculation)

prepare_for_submission(*tempfolder*)

Create the input files from the input nodes passed to this instance of the *CalcJob*.

Parameters **tempfolder** – an *aiida.common.folders.Folder* to temporarily write files on disk

Returns *aiida.common.datastructures.CalcInfo* instance

KKRcode

Input plug-in for a KKR calculation.

class aiida_kkr.calculations.kkr.**KkrCalculation**(**args*, ***kwargs*)

AiiDA calculation plugin for a KKR calculation.

_kick_out_corestates_kkrhost(*local_copy_list*, *tempfolder*)

Compare value of core states from potential file in local_copy_list with EMIN and kick corestate out of potential if they lie inside the energy contour.

_prepare_qdos_calc(*parameters*, *kpath*, *structure*, *tempfolder*, *use_alat_input*)

prepare a qdos (i.e. bandstructure) calculation, can only be done if k-points are given in input Note: this changes some settings in the parameters to ensure a DOS contour and low smearing temperature Also the qvec.dat file is written here.

_set_ef_value_potential(*ef_set*, *local_copy_list*, *tempfolder*)
Set EF value ef_set in the potential file.

_set_parent_remotedata(*remotedata*)
Used to set a parent remotefolder in the restart of fleur.

_use_decimation(*parameters*, *tempfolder*)
Activate decimation mode and copy decifile from output of deciout_parent calculation

_use_initial_noco_angles(*parameters*, *structure*, *tempfolder*)
Set starting values for non-collinear calculation (writes nonco_angle.dat to tempfolder). Adapt FIXMOM runopt according to fix_dir input in initial_noco_angle input node

classmethod define(*spec*)
Init internal parameters at class load time

prepare_for_submission(*tempfolder*)
Create input files.

param tempfolder aiida.common.folders.Folder subclass where the plugin should put all its files.

param inputdict dictionary of the input nodes as they would be returned by get_inputs_dict

aiida_kkr.calculations.kkr.**_update_params**(*parameters*, *change_values*)
change parameters node from change_values list of key value pairs Retrun input parameter node if change_values list is empty

KKRcode - calculation importer

Plug-in to import a KKR calculation. This is based on the PwImmigrantCalculation of the aiida-quantumespresso plugin.

class aiida_kkr.calculations.kkrimporter.**KkrImporterCalculation**(**args*, ***kwargs*)
Importer dummy calculation for a previous KKR run

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_names** – The file names of the job's input file.
- **output_file_name** (*dict with str entries*) – The file names of the job's output file (i.e. the file containing the stdout of kkr.x).

_init_internal_params()
Init internal parameters at class load time

KKRimp

Input plug-in for a KKRimp calculation.

class aiida_kkr.calculations.kkrimp.**KkrimpCalculation**(**args*, ***kwargs*)
AiiDA calculation plugin for a KKRimp calculation.

_change_atominfo(*imp_info*, *kkrflex_file_paths*, *tempfolder*)
change kkrflex_atominfo to match impurity case

_check_and_extract_input_nodes(*tempfolder*)
Extract input nodes from inputdict and check consitency of input nodes :param inputdict: dict of inputnodes :returns:

- parameters (aiida_kkr.tools.kkr_params.kkrparams), optional: parameters of KKRimp that end up in config.cfg
- code (KKRimpCodeNode): code of KKRimp on some machine
- imp_info (DictNode): parameter node of the impurity information, extracted from host_parent_calc
- kkrflex_file_paths (dict): dictionary of {filenames: absolute_path_to_file} for the kkrflex-files
- shapfun_path (str): absolute path of the shapefunction of the host parent calculation
- host_parent_calc (KkrCalculation): node of the parent host calculation where the kkrflex-files were created
- impurity_potential (SinglefileData): single file data node containing the starting potential for the impurity calculation
- parent_calc_folder (RemoteData): remote directory of a parent KKRimp calculation

_check_key_setting_consistency(*params_kkrimp*, *key*, *val*)
Check if key/value pair that is supposed to be set is not in conflict with previous settings of parameters in params_kkrimp

_extract_and_write_config(*parent_calc_folder*, *params_host*, *parameters*, *tempfolder*, *GFhost_folder*)
fill kkr params for KKRimp and write config file also writes kkrflex_llyfac file if Lloyd is used in the host system

_get_and_verify_hostfiles(*tempfolder*)
Check inputdict for host_Greenfunction_folder and extract impurity_info, paths to kkrflex-files and path of shapefun file

Parameters **inputdict** – input dictionary containing all input nodes to KkrimpCalculation

Returns

- imp_info: Dict node containing impurity information like position, Z_imp, cluster size, etc.
- kkrflex_file_paths: dict of absolute file paths for the kkrflex files
- shapefun_path: absolute path of the shapefunction file in the host calculation (needed to construct shapefun_imp)
- shapes: mapping array of atoms to shapes (<SHAPE> input)

Note shapefun_path is None if host_Greenfunction calculation was not full-potential

Raises

- InputValidationError, if inputdict does not contain 'host_Greenfunction'
- InputValidationError, if host_Greenfunction_folder not of right type
- UniquenessError, if host_Greenfunction_folder does not have exactly one parent
- InputValidationError, if host_Greenfunction does not have an input node impurity_info
- InputValidationError, if host_Greenfunction was not a KKRFLEX calculation

_get_pot_and_shape(*imp_info*, *shapefun*, *shapes*, *impurity_potential*, *parent_calc_folder*, *tempfolder*, *structure*)
write shapefun from impurity info and host shapefun and copy imp. potential

returns: file handle to potential file

adapt_retrieve_tmatnew(*tempfolder*, *allopts*, *retrieve_list*)
Add out_magneticmoments and orbitalmoments files to retrieve list

add_jij_files(*tempfolder*, *retrieve_list*)
check if KkrimpCalculation is in Jij mode and add OUT_JIJMAT to retrieve list if needed

add_lmdos_files_to_retrieve(*tempfolder*, *allopts*, *retrieve_list*, *kkrflex_file_paths*)
Add DOS files to retrieve list

create_or_update_ldaupot(*parent_calc_folder*, *tempfolder*)
Writes ldaupot to tempfolder.

If parent_calc_folder is found and it contains an onld ldaupot, we reuse the values for wldau, uldau and phi from there.

classmethod define(*spec*)
Init internal parameters at class load time

classmethod get_ldaupot_from_retrieved(*retrieved*, *tempfolder*)
Extract ldaupot from output of KKRimp retreived to tempfolder. The extracted file in tempfolder will be named ldaupot_old.

returns True of ldaupot was found, otherwise returns False

get_old_ldaupot(*parent_calc_folder*, *tempfolder*)
Copy old ldaupot from retrieved of parent or extract from tarball. If no parent_calc_folder is present this step is skipped.

get_remote_symlink(*local_copy_list*)
Check if host GF is found on remote machine and reuse from there

get_run_test_opts(*parameters*)
Extract run and test options from input parameters

init_ldau(*tempfolder*, *retrieve_list*, *parent_calc_folder*)
Check if settings_LDAU is in input and set up LDA+U calculation. Reuse old ldaupot of parent_folder contains a file ldaupot.

prepare_for_submission(*tempfolder*)
Create input files.

param tempfolder aiida.common.folders.Folder subclass where the plugin should put all its files.

param inputdict dictionary of the input nodes as they would be returned by get_inputs_dict

aiida_kkr.calculations.kkrimp.**get_ldaupot_text**(*ldau_settings*, *ef_Ry*, *natom*, *initialize=True*)
create the text for the ldaupot file

Workflows

This section describes the aiida-kkr workflows.

Generate KKR start potential

In this module you find the base workflow for a dos calculation and some helper methods to do so with AiiDA

class aiida_kkr.workflows.voro_start.**kkr_startpot_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

Workchain create starting potential for a KKR calculation by running voronoi and getting the starting DOS for first checks on the validity of the input setting. Starts from a structure together with a KKR parameter node.

Parameters

- **wf_parameters** – (Dict), Workchain specifications
- **options** – (Dict), specifications for the computer
- **structure** – (StructureData), aiida structure node to begin calculation from (needs to contain vacancies, if KKR needs empty spheres)
- **kkr** – (Code)
- **voronoi** – (Code)
- **calc_parameters** – (Dict), KKR parameter set, passed on to voronoi run.

Return result_kkr_startpot_wc (Dict), Information of workflow results like Success, last result node, dos array data

check_dos()
checks if dos of starting potential is ok

check_voronoi()
check voronoi output. return True/False if voronoi output is ok/problematic if output is problematic try to increase some parameters (e.g. cluster radius) and rerun up tp N_rerun_max times initializes with returning True

classmethod define(*spec*)
Defines the outline of the workflow.

do_iteration_check()
check if another iteration should be done

error_handler()
Capture errors raised in validate_input

find_cluster_radius_alat()
Find an estimate for the cluster radius that comes close to having nclsmin atoms in the cluster.

get_dos()
call to dos sub workflow passing the appropriate input and submitting the calculation

classmethod get_wf_defaults(*silent=False*)
Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters. returns _wf_defaults

return_results()
return the results of the dos calculations This should run through and produce output nodes even if everything failed, therefore it only uses results from context.

run_voronoi()
run voronoi calculation with parameters from input

start()
init context and some parameters

aiida_kkr.workflows.voro_start.**update_voro_input**(*params_old*, *updatenode*, *voro_output*)

Pseudo wf used to keep track of updated parameters in voronoi calculation. voro_output only enters as dummy argument for correct connection but logic using this value is done somewhere else.

KKR scf cycle

In this module you find the base workflow for converging a kkr calculation and some helper methods to do so with AiiDA

aiida_kkr.workflows.kkr_scf.**create_scf_result_node**(***kwargs*)

This is a pseudo wf, to create the right graph structure of AiiDA. This workfunction will create the output node in the database. It also connects the output_node to all nodes the information commes from. So far it is just also parsed in as argument, because so far we are to lazy to put most of the code overworked from return_results in here.

aiida_kkr.workflows.kkr_scf.**extract_noco_angles**(***kwargs*)

Extract noco angles from retrieved nonco_angles_out.dat files and save as Dict node which can be used as initial values for the next KkrCalculation. New angles are compared to old angles and if they are closer thanfix_dir_threshold they are not allowed to change anymore

aiida_kkr.workflows.kkr_scf.**get_site_symbols**(*structure*)

extract the site number taking into account a possible CPA structure

class aiida_kkr.workflows.kkr_scf.**kkr_scf_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

Workchain for converging a KKR calculation (SCF).

It converges the charge potential. Two paths are possible:

(1) Start from a structure and run a voronoi calculation first, optional with calc_parameters (2) Start from an existing Voronoi or KKR calculation, with a remoteData

Parameters

- **wf_parameters** – (Dict), Workchain Specifications
- **options** – (Dict); specifications for the computer
- **structure** – (StructureData), Crystal structure
- **calc_parameters** – (Dict), Voronoi/Kkr Parameters
- **remote_data** – (RemoteData), from a KKR, or Voronoi calculation
- **voronoi** – (Code)
- **kkr** – (Code)

Return output_kkr_scf_wc_para (Dict), Information of workflow results like Success, last result node, list with convergence behavior

minimum input example: 1. Code1, Code2, Structure, (Parameters), (wf_parameters) 2. Code2, remote_data, (Parameters), (wf_parameters)

maximum input example: 1. Code1, Code2, Structure, Parameters

wf_parameters: {'queue_name' [String,] 'resources' : dict({"num_machines": int, "num_mpiprocs_per_machine" : int}) 'walltime' : int}

2. Code2, (remote-data), wf_parameters as in 1.

Hints: 1. This workflow does not work with local codes!

_get_new_noco_angles()
extract nonco angles from output of calculation, if fix_dir is True we skip this and leave the initial angles unchanged Here we update self.ctx.initial_noco_angles with the new values

check_dos()
checks if dos of final potential is ok

check_input_params(*params*, *is_voronoi=False*)
Checks input parameter consistency and aborts wf if check fails.

check_voronoi()
check output of kkr_startpot_wc workflow that creates starting potential, shapefun etc.

condition()
check convergence condition

convergence_on_track()
Check if convergence behavior of the last calculation is on track (i.e. going down)

classmethod define(*spec*)
Defines the outline of the workflow.

get_dos()
call to dos sub workflow passing the appropriate input and submitting the calculation

classmethod get_wf_defaults(*silent=False*)
Print and return _wf_default dictionary. Can be used to easily create set of wf_parameters. returns _wf_default, _options_default

inspect_kkr()
check for convergence and store some of the results of the last calculation to context

return_results()
return the results of the calculations This shoudl run through and produce output nodes even if everything failed, therefore it only uses results from context.

run_kkr()
submit a KKR calculation

run_voronoi()
run the voronoi step calling voro_start workflow

start()
init context and some parameters

update_kkr_params()
update set of KKR parameters (check for reduced mixing, change of mixing strategy, change of accuracy setting)

validate_input()
validate input and find out which path (1, or 2) to take # return True means run voronoi if false run kkr directly

Density of states

In this module you find the base workflow for a dos calculation and some helper methods to do so with AiiDA

class aiida_kkr.workflows.dos.**kkr_dos_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

Workchain a DOS calculation with KKR starting from the remoteData node of a previous calculation (either Voronoi or KKR).

Parameters

- **wf_parameters** – (Dict); Workchain specifications
- **options** – (Dict); specifications for the computer
- **remote_data** – (RemoteData), mandatory; from a KKR or Vornoi calculation
- **kkr** – (Code), mandatory; KKR code running the dos calculation

Return result_kkr_dos_wc (Dict), Information of workflow results like Success, last result node, list with convergence behavior

classmethod define(*spec*)

Defines the outline of the workflow.

get_dos()

submit a dos calculation and interpolate result if returns complete

classmethod get_wf_defaults(*silent=False*)

Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters. returns _wf_defaults

return_results()

Collect results, parse DOS output and link output nodes to workflow node

set_params_dos()

take input parameter node and change to DOS contour according to input from wf_parameter input internally calls the update_params work function to keep track of provenance

start()

init context and some parameters

validate_input()

validate input and find out which path (1, or 2) to take # return True means run voronoi if false run kkr directly

aiida_kkr.workflows.dos.**parse_dosfiles**(*dos_retrieved*)

parse dos files to XyData nodes

Bandstructure

This module contains the band structure workflow for KKR which is done by calculating the k-resolved spectral density also known as Bloch spectral function.

class aiida_kkr.workflows.bs.**kkr_bs_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

Workchain for BandStructure calculation, starting from RemoteFolderData of the previous converged KKR calculation remote folder data

inputs: :param wf_parameters: (Dict), (optional); Workchain Specifications, contains nepts, tempr, emin (in eV relative to EF), emax (in eV),

and RCLUSTZ (can be used to increase the screening cluster radius) keys.

Parameters

- **options** – (Dict), (optional); Computer Specifications, scheduler command, parallel or serial
- **kpoints** – (KpointsData),(optional); Kpoints data type from the structure, but not mendatory as it can be extracted from structure internaly from the remote data
- **remote_data** – (RemoteData)(mendaory); From the previous kkr-converged calculation.
- **kkr** – (Code)(mendaory); KKR code specifiaction
- **label** – (Str) (optional) ; label for WC but will be found in the "result_wf" output Dict as 'BS_wf_label' key
- **description** – (Str) (optional) : description for WC but will be found in the "result_wf" output Dict as 'BS_wf_description' key

returns: :out BS_Data : (ArrayData) ; Consist of BlochSpectralFunction, k_points (list), energy_points (list), special_kpoints(dict) :out result_wf: (Dict); work_chain_specifications node, BS_data node, remote_folder node

classmethod define(*spec*)
Layout of the workflow, defines the input nodes and the outline of the workchain

get_BS()
submit the KkrCalcultion with the qdos settings for a bandstructure calculation

classmethod get_wf_defaults(*silent=False*)
Return the default values of the workflow parameters (wf_parameters input node)

return_results()
Collect results, parse BS_calc output and link output nodes to workflow node

set_params_BS()
set kkr parameters for the bandstructure (i.e. qdos) calculation

start()
set up context of the workflow

validate_input()
validate input and find out which path (converged kkr calc or wf) to take return True means run voronoi if false run kkr directly

aiida_kkr.workflows.bs.**parse_BS_data**(*retrieved_folder*, *fermi_level*, *kpoints*)
parse the qdos files from the retreived folderand save as ArrayData

aiida_kkr.workflows.bs.**set_energy_params**(*econt_new*, *ef*, *para_check*)
set energy contour values to para_check internally convert from relative eV units to absolute Ry units

Equation of states

In this module you find the base workflow for a EOS calculation and some helper methods to do so with AiiDA

aiida_kkr.workflows.eos.**get_primitive_structure**(*structure*, *return_all*)
calls get_explicit_kpoints_path which gives primitive structure auxiliary workfunction to keep provenance

class aiida_kkr.workflows.eos.**kkr_eos_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Workchain of an equation of states calculation with KKR.

Layout of the workflow:

1. determine V0, scale_range, etc. from input

2. **run voro_start for V0 and smallest volume** 2.1 get minimum for RMTCORE (needs to be fixed for all calculations to be able to compare total energies
3. submit kkr_scf calculations for all volumes using RMTCORE setting determined in step 2
4. collect results

check_voro_out()
check outout of vorostart workflow and create input for rest of calculations (rmtcore setting etc.)

collect_data_and_fit()
collect output of KKR calculations and perform eos fitting to collect results

classmethod define(*spec*)
Defines the outline of the workflow.

classmethod get_wf_defaults(*silent=False*)
Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters. returns _wf_defaults, _options_default

prepare_strucs()
create new set of scaled structures using the 'rescale' workfunction (see end of the workflow)

return_results()
create output dictionary and run output node generation

run_kkr_steps()
submit KKR calculations for all structures, skip vorostart step for smallest structure

run_vorostart()
run vorostart workflow for smallest structure to determine rmtcore setting for all others

start()
initialize context and check input nodes

aiida_kkr.workflows.eos.**rescale**(*inp_structure*, *scale*)
Rescales a crystal structure. Keeps the provanance in the database.

:param inp_structure, a StructureData node (pk, or uuid) :param scale, float scaling factor for the cell

Returns New StrcutureData node with rescalled structure, which is linked to input Structure and None if inp_structure was not a StructureData

copied and modified from aiida_fleur.tools.StructureData_util

aiida_kkr.workflows.eos.**rescale_no_wf**(*structure*, *scale*)
Rescales a crystal structure. DOES NOT keep the provanence in the database.

:param structure, a StructureData node (pk, or uuid) :param scale, float scaling factor for the cell

Returns New StrcutureData node with rescalled structure, which is linked to input Structure and None if inp_structure was not a StructureData

copied and modified from aiida_fleur.tools.StructureData_util

Find Green Function writeout for KKRimp

In this module you find the base workflow for writing out the kkr_flexfiles and some helper methods to do so with AiiDA

class aiida_kkr.workflows.gf_writeout.**kkr_flex_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)

Workchain of a kkr_flex calculation to calculate the Green function with KKR starting from the RemoteData node of a previous calculation (either Voronoi or KKR).

Parameters

- **options** – (Dict), Workchain specifications
- **wf_parameters** – (Dict), Workflow parameters that deviate from previous KKR RemoteData
- **remote_data** – (RemoteData), mandatory; from a converged KKR calculation
- **kkr** – (Code), mandatory; KKR code running the flexfile writeout
- **impurity_info** – Dict, mandatory: node specifying information of the impurities in the system

Return workflow_info (Dict), Information of workflow results like success, last result node, list with convergence behavior

Return GF_host_remote (RemoteData), host GF of the system

classmethod define(*spec*)
Defines the outline of the workflow

get_flex()
Submit a KKRFLEX calculation

classmethod get_wf_defaults()
Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters. returns _wf_defaults

move_kkrflex_files()
Move the kkrflex files from the remote folder to KkrimpCalculation._DIRNAME_GF_UPLOAD on the remote computer's working dir. This skips retrieval to the file repository and reduces cluttering the database.

return_results()
Return the results of the KKRFLEX calculation. This should run through and produce output nodes even if everything failed, therefore it only uses results from context.

set_params_flex()
Take input parameter node and change to input from wf_parameter and options

start()
init context and some parameters

validate_input()
Validate input

KKRimp self-consistency

In this module you find the sub workflow for the kkrimp self consistency cycle and some helper methods to do so with AiiDA

aiida_kkr.workflows.kkr_imp_sub.**clean_raw_input**(*successful*, *pks_calcs*, *dry_run=False*)
Clean raw_input directories that contain copies of shapefun and potential files This however breaks provenance (strictly speaking) and therefore should only be done for the calculations of a successfully finished workflow (see email on mailing list from 25.11.2019).

aiida_kkr.workflows.kkr_imp_sub.**clean_sfd**(*sfd_to_clean*, *nkeep=30*)
Clean up potential file (keep only header) to save space in the repository WARNING: this breaks cachability!

aiida_kkr.workflows.kkr_imp_sub.**extract_imp_pot_sfd**(*retrieved_folder*)
Extract potential file from retrieved folder and save as SingleFileData

class aiida_kkr.workflows.kkr_imp_sub.**kkr_imp_sub_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Workchain of a kkrimp self consistency calculation starting from the host-impurity potential of the system. (Not the entire kkr_imp workflow!)

Parameters

- **options** – (Dict), Workchain specifications
- **wf_parameters** – (Dict), specifications for the calculation
- **host_imp_startpot** – (RemoteData), mandatory; input host-impurity potential
- **kkrimp** – (Code), mandatory; KKRimp code converging the host-imp-potential
- **remote_data** – (RemoteData), mandatory; remote folder of a previous kkrflex calculation containing the flexfiles . . .
- **kkrimp_remote** – (RemoteData), remote folder of a previous kkrimp calculation
- **impurity_info** – (Dict), Parameter node with information about the impurity cluster

Return workflow_info (Dict), Information of workflow results like success, last result node, list with convergence behavior

Return host_imp_pot (SinglefileData), output potential of the sytem

condition()
check convergence condition

convergence_on_track()
Check if convergence behavior of the last calculation is on track (i.e. going down)

classmethod define(*spec*)
Defines the outline of the workflow

error_handler()
Capture errors raised in validate_input

classmethod get_wf_defaults(*silent=False*)
Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters.

returns _wf_defaults

inspect_kkrimp()
check for convergence and store some of the results of the last calculation to context

return_results()
Return the results of the calculations This should run through and produce output nodes even if everything failed, therefore it only uses results from context.

run_kkrimp()
submit a KKR impurity calculation

start()
init context and some parameters

update_kkrimp_params()
update set of KKR parameters (check for reduced mixing, change of mixing strategy, change of accuracy setting)

validate_input()
validate input and catch possible errors from the input

aiida_kkr.workflows.kkr_imp_sub.**remove_out_pot_impcalcs**(*successful*, *pks_all_calcs*, *dry_run=False*)
Remove out_potential file from all but the last KKRimp calculation if workflow was successful

Usage:

```
imp_wf = load_node(266885) # maybe start with outer workflow
pk_imp_scf = imp_wf.outputs.workflow_info['used_subworkflows'].get('kkr_imp_sub')
imp_scf_wf = load_node(pk_imp_scf) # this is now the imp scf sub workflow
successful = imp_scf_wf.outputs.workflow_info['successful']
pks_all_calcs = imp_scf_wf.outputs.workflow_info['pks_all_calcs']
```

KKRimp complete calculation

In this module you find the total workflow for a kkr impurity calculation and some helper methods to do so with AiiDA

class aiida_kkr.workflows.kkr_imp.**kkr_imp_wc**(*inputs=None*, *logger=None*, *runner=None*, *enable_persistence=True*)
Workchain of a kkrimp calculation starting either from scratch (with a structure and impurity_info node), or with a converged host potential and impurity startpotentials, ... to calculate the converged host-impurity potential of the system.

Parameters

- **options** – (Dict), Workchain specifications
- **wf_parameters** – (Dict), specifications for the kkr impurity workflow
- **voro_aux_parameters** – (Dict), specification for the auxiliary voronoi calculation for the impurity
- **kkrimp** – (Code), mandatory: KKRimp code converging the host-imp-potential
- **kkr** – (Code), mandatory: KKR code for calculation the host potential
- **voronoi** – (Code), mandatory: Voronoi code to generate the impurity startpot
- **remote_data_gf** – (RemoteData): remote folder of a previous kkrflex calculation containing the flexfiles ...
- **remote_data_host** – (RemoteData): remote folder of a converged KKR host calculation

Return workflow_info (Dict), Information of workflow results

Return last_calc_output_parameters (Dict), output parameters of the last called calculation

Return last_calc_info (Dict), information of the last called calculation

construct_startpot()
Take the output of GF writeout and the converged host potential as well as the auxiliary startpotentials for the impurity to construct the startpotential for the KKR impurity sub workflow

classmethod define(*spec*)
Defines the outline of the workflow

final_cleanup()
Remove unneeded files to save space

get_ef_from_parent()
Extract Fermi level in Ry to which starting potential is set

classmethod get_wf_defaults(*silent=False*)
Print and return _wf_defaults dictionary. Can be used to easily create set of wf_parameters.

returns _wf_defaults

has_starting_potential_input()
check whether or not a starting potential needs to be created

return_results()
Return the results and create all of the output nodes

run_gf_writeout()
Run the gf_writeout workflow to calculate the host Green's function and the KKR flexfiles using the converged host remote folder and the impurity info node

run_kkrimp_scf()
Uses both the previously generated host-impurity startpotential and the output from the GF writeout workflow as inputs to run the kkrimp_sub workflow in order to converge the host-impurity potential

run_voroaux()
Perform a voronoi calculation for every impurity charge using the structure from the converged KKR host calculation

start()
Init context and some parameters

validate_input()
Validate the input and catch possible errors from the input

Calculation parsers

This section describes the different parsers classes for calculations.

Voronoi Parser

class aiida_kkr.parsers.voro.**VoronoiParser**(*calc*)
Parser class for parsing output of voronoi code..

__init__(*calc*)
Initialize the instance of Voronoi_Parser

parse(*debug=False*, ***kwargs*)
Parse output data folder, store results in database.

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

Returns nothing if everything is fine or an exit code defined in the voronoi calculation class

KKRcode Parser

Parser for the KKR Code. The parser should never fail, but it should catch all errors and warnings and show them to the user.

class aiida_kkr.parsers.kkr.**KkrParser**(*calc*)
Parser class for parsing output of KKR code..

__init__(*calc*)
Initialize the instance of KkrParser

parse(*debug=False*, ***kwargs*)
Parse output data folder, 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)`)

remove_unnecessary_files()
Remove files that are not needed anymore after parsing The information is completely parsed (i.e. in outdict of calculation) and keeping the file would just be a duplication.

KKRcode - calculation importer Parser

Parser for the KKR imprter, slight modification to KKr parser (dealing of missing output files). The parser should never fail, but it should catch all errors and warnings and show them to the user.

class aiida_kkr.parsers.kkrimporter.**KkrImporterParser**(*calc*)
Parser class for parsing output of KKR code after import

__init__(*calc*)
Initialize the instance of KkrParser

KKRimp Parser

Parser for the KKR-impurity Code. The parser should never fail, but it should catch all errors and warnings and show them to the user.

class aiida_kkr.parsers.kkrimp.**KkrimpParser**(*calc*)
Parser class for parsing output of the KKRimp code..

__init__(*calc*)
Initialize the instance of KkrimpParser

cleanup_outfiles(*fileidentifier*, *keyslist*)
open file and remove unneeded output

final_cleanup()
Create a tarball of the rest.

parse(*debug=False*, ***kwargs*)
Parse output data folder, store results in database.

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

remove_unnecessary_files()
Remove files that are not needed anymore after parsing The information is completely parsed (i.e. in outdict of calculation) and keeping the file would just be a duplication.

Voronoi Parser

class aiida_kkr.parsers.voro.**VoronoiParser**(*calc*)
Parser class for parsing output of voronoi code..

__init__(*calc*)
Initialize the instance of Voronoi_Parser

parse(*debug=False*, ***kwargs*)
Parse output data folder, store results in database.

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

Returns nothing if everything is fine or an exit code defined in the voronoi calculation class

Tools

Here the tools provided by aiida_kkr are described.

Common (work)functions that need aiida

Here workfunctions and normal functions using aiida-stuff (typically used within workfunctions) are collected.

aiida_kkr.tools.common_workfunctions.**check_2Dinput_consistency**(*structure*, *parameters*)
Check if structure and parameter data are complete and matching.

Parameters

- **input** – structure, needs to be a valid aiida StructureData node
- **input** – parameters, needs to be valid aiida Dict node

returns (False, errormessage) if an inconsistency has been found, otherwise return (True, '2D consistency check complete')

aiida_kkr.tools.common_workfunctions.**extract_potname_from_remote**(*parent_calc_folder*)
extract the bname of the output potential from a RemoteData folder

aiida_kkr.tools.common_workfunctions.**find_cluster_radius**(*structure*, *nclsmin*, *n_max_box=50*, *nbins=100*)
Takes structure information (cell and site positions) and computes the minimal cluster radius needed such that all clusters around all atoms contain more than *nclsmin* atoms.

Note Here we assume spherical clusters around the atoms!

Parameters

- **structure** – input structure for which the clusters are analyzed
- **nclsmin** – minimal number of atoms in the screening cluster
- **n_max_box** – maximal number of supercells in 3D volume

- **nbins** – number of bins in which the cluster number is analyzed

Returns minimal cluster radius needed in Angstroem

Returns minimal cluster radius needed in units of the lattice constant

aiida_kkr.tools.common_workfunctions.**generate_inputcard_from_structure**(*parameters*, *structure*, *input_filename*, *parent_calc=None*, *shapes=None*, *isvoronoi=False*, *use_input_alat=False*, *vca_structure=False*)

Takes information from parameter and structure data and writes input file 'input_filename'

Parameters

- **parameters** – input parameters node containing KKR-related input parameter
- **structure** – input structure node containing lattice information
- **input_filename** – input filename, typically called 'inputcard'

optional arguments :param parent_calc: input parent calculation node used to determine if EMIN

parameter is automatically overwritten (from voronoi output) or not

Parameters

- **shapes** – input shapes array (set automatically by aiida_kkr.calculations.Kkrcalculation and shall not be overwritten)
- **isvoronoi** – tell whether or not the parameter set is for a voronoi calculation or kkr calculation (have different lists of mandatory keys)
- **use_input_alat** – True/False, determines whether the input alat value is taken or the new alat is computed from the Bravais vectors

Note assumes valid structure and parameters, i.e. for 2D case all necessary information has to be given. This is checked with function 'check_2D_input' called in aiida_kkr.calculations.Kkrcalculation

aiida_kkr.tools.common_workfunctions.**get_inputs_common**(*calculation*, *code*, *remote*, *structure*, *options*, *label*, *description*, *params*, *serial*, *imp_info=None*, *host_GF=None*, *imp_pot=None*, *kkrimp_remote=None*, *host_GF_Efshift=None*, ***kwargs*)

Base function common in get_inputs_* functions for different codes

aiida_kkr.tools.common_workfunctions.**get_inputs_kkr**(*code*, *remote*, *options*, *label=''*, *description=''*, *parameters=None*, *serial=False*, *imp_info=None*)

Get the input for a voronoi calc. Wrapper for KkrProcess setting structure, code, options, label, description etc. :param code: a valid KKRcode installation (e.g. input from Code.get_from_string('codename@computername')) :param remote: remote directory of parent calculation (Voronoi or previous KKR calculation)

aiida_kkr.tools.common_workfunctions.**get_inputs_kkrimp**(*code*, *options*, *label=''*, *description=''*, *parameters=None*, *serial=False*, *imp_info=None*, *host_GF=None*, *imp_pot=None*, *kkrimp_remote=None*, *host_GF_Efshift=None*)

Get the input for a kkrimp calc. Wrapper for KkrimpProcess setting structure, code, options, label, description etc. :param code: a valid KKRimpcode installation (e.g. input from Code.get_from_string('codename@computername')) TBD

aiida_kkr.tools.common_workfunctions.**get_inputs_kkrimporter**(*code*, *remote*, *options*, *label=''*, *description=''*, *parameters=None*, *serial=False*)

Get the input for a voronoi calc. Wrapper for KkrProcess setting structure, code, options, label, description etc.

aiida_kkr.tools.common_workfunctions.**get_inputs_voronoi**(*code*, *structure*, *options*, *label=''*, *description=''*, *params=None*, *serial=True*, *parent_KKR=None*)

Get the input for a voronoi calc. Wrapper for VoronoiProcess setting structure, code, options, label, description etc.

aiida_kkr.tools.common_workfunctions.**get_parent_paranode**(*remote_data*)

Return the input parameter of the parent calculation giving the remote_data node

aiida_kkr.tools.common_workfunctions.**kick_out_corestates**(*potfile*, *potfile_out*, *emin*)

Read potential file and kick out all core states that lie higher than emin. If no core state lies higher than emin then the output potential will be the same as the input potential :param potfile: input potential :param potfile_out: output potential where some core states are kicked out :param emin: minimal energy above which all core states are kicked out from potential :returns: number of lines that have been deleted

aiida_kkr.tools.common_workfunctions.**kick_out_corestates_wf**(*potential_sfd*, *emin*)

Workfunction that kicks out all core states from single file data potential that are higher than emin. :param potential_sfd: SinglefileData type of potential :param emin: Energy threshold above which all core states are removed from potential (Float) :returns: potential without core states higher than emin (SinglefileData)

aiida_kkr.tools.common_workfunctions.**neworder_potential_wf**(*settings_node*, *parent_calc_folder*, ***kwargs*)

Workfunction to create database structure for aiida_kkr.tools.modify_potential.neworder_potential function A temporary file is written in a Sandbox folder on the computer specified via the input computer node before the output potential is stored as SinglefileData in the Database.

Parameters

- **settings_node** – settings for the neworder_potential function (Dict)
- **parent_calc_folder** – parent calculation remote folder node where the input potential is retreived from (RemoteData)

- **parent_calc_folder2** – *optional*, parent calculation remote folder node where the second input potential is retreived from in case 'pot2' and 'replace_newpos' are also set in settings_node (RemoteData)
- **debug** – *optional*, contol wether or not debug information is written out (aiida.orm.Bool)

Returns output_potential node (SinglefileData)

Note: The settings_node dictionary needs to be of the following form:

```
settings_dict = {'neworder': [list of intended order in output potential]}
```

Optional entries are:

```
'out_pot': '<filename_output_potential>'  name of the output potential file,␣
↪defaults to 'potential_neworder' if not specified
'pot1': '<filename_input_potential>'      if not given we will try to find it␣
↪from the type of the parent remote folder
'pot2': '<filename_second_input_file>'
'replace_newpos': [[position in neworder list which is replace with potential␣
↪from pot2, position in pot2 that is chosen for replacement]]
'switch_spins': [indices of atom for which spins are exchanged] (indices refer to␣
↪position in neworder input list)
'label': 'label_for_output_node'
'description': 'longer_description_for_output_node'
```

---

aiida_kkr.tools.common_workfunctions.**structure_from_params**(*parameters*)

Construct aiida structure out of kkr parameter set (if ALATBASIS, RBASIS, ZATOM etc. are given)

Parameters **input** – parameters, kkrparams object with structure information set (e.g. extracted from read_inputcard function)

Returns success, boolean to determine if structure creatoin was successful

Returns structure, an aiida StructureData object

aiida_kkr.tools.common_workfunctions.**test_and_get_codenode**(*codenode*, *expected_code_type*, *use_exceptions=False*)

Pass a code node and an expected code (plugin) type. Check that the code exists, is unique, and return the Code object.

Parameters

- **codenode** – the name of the code to load (in the form label@machine)
- **expected_code_type** – a string with the plugin that is expected to be loaded. In case no plugins exist with the given name, show all existing plugins of that type
- **use_exceptions** – if True, raise a ValueError exception instead of calling sys.exit(1)

Returns a Code object

Example usage from kkr_scf workflow:

if 'voronoi' in inputs:

try: test_and_get_codenode(inputs.voronoi, 'kkr.voro', use_exceptions=True)

except ValueError:

error = ("The code you provided for voronoi does not " "use the plugin kkr.voro")

self.control_end_wc(error)

aiida_kkr.tools.common_workfunctions.**update_params**(*node*, *nodename=None*, *nodedesc=None*, ***kwargs*)

Update parameter node given with the values given as kwargs. Returns new node.

Parameters

- **node** – Input parameter node (needs to be valid KKR input parameter node).
- ****kwargs** – Input keys with values as in kkrparams.
- **linkname** – Input linkname string. Give link from old to new node a name . If no linkname is given linkname defaults to 'updated parameters'

Returns parameter node

Example usage OutputNode = KkrCalculation.update_params(InputNode, EMIN=-1, NSTEPS=30)

Note Keys are set as in kkrparams class. Check documentation of kkrparams for further information.

Note If kwargs contain the key *add_direct*, then no kkrparams instance is used and no checks are performed but the dictionary is filled directly!

Note By default nodename is 'updated KKR parameters' and description contains list of changed

aiida_kkr.tools.common_workfunctions.**update_params_wf**(*parameternode*, *updatenode*, ***link_inputs*)

Work function to update a KKR input parameter node. Stores new node in database and creates a link from old parameter node to new node Returns updated parameter node using update_params function

Note Input nodes need to be valid aiida Dict objects.

Parameters

- **parameternode** – Input aiida Dict node cotaining KKR specific parameters
- **updatenode** – Input aiida Dict node containing a dictionary with the parameters that are supposed to be changed.

Note If 'nodename' is contained in dict of updatenode the string corresponding to this key will be used as nodename for the new node. Otherwise a default name is used

Note Similar for 'nodedesc' which gives new node a description

Example updated_params = Dict(dict={'nodename': 'my_changed_name', 'nodedesc': 'My description text', 'EMIN': -1, 'RMAX': 10.}) new_params_node = update_params_wf(input_node, updated_params)

aiida_kkr.tools.common_workfunctions.**vca_check**(*structure*, *parameters*)

KKRimp tools

Tools for the impurity caluclation plugin and its workflows

aiida_kkr.tools.tools_kkrimp.**create_scoef_array**(*structure, radius, h=-1, vector=[0.0, 0.0, 1.0], i=0, alat_input=None*)

Creates the arrays that should be written into the 'scoef' file for a certain structure. Needed to conduct an impurity KKR calculation.

Parameters

- **structure** – input structure of the StructureData type.

- **radius** – input cutoff radius in Ang. units.
- **h** – height of the cutoff cylinder (negative for spherical cluster shape). For negative values, clust_shape will be automatically assumed as 'spherical'. If there will be given a h > 0, the clust_shape will be 'cylindrical'.
- **vector** – orientation vector of the cylinder (just for clust_shape='cylindrical').
- **i** – atom index around which the cluster should be centered. Default: 0 (first atom in the structure).
- **alat_input** – input lattice constant in Ang. If *None* use the lattice constant that is automatically found. Otherwise rescale everything.

aiida_kkr.tools.tools_kkrimp.**find_neighbors**(*structure, structure_array, i, radius, clust_shape='spherical', h=0.0, vector=[0.0, 0.0, 1.0]*)

Applies periodic boundary conditions and obtains the distances between the selected atom i in the cell and all other atoms that lie within a cutoff radius r_cut. Afterwards an numpy array with all those atoms including atom i (x_res) will be returned.

Parameters

- **structure** – input parameter of the StructureData type containing the three bravais lattice cell vectors
- **structure_array** – input numpy structure array containing all the structure related data
- **i** – centered atom at which the origin lies (same one as in select_reference)
- **radius** – Specifies the radius of the cylinder or of the sphere, depending on clust_shape. Input in units of the lattice constant.
- **clust_shape** – specifies the shape of the cluster that is used to determine the neighbors for the 'scoef' file. Default value is 'spherical'. Other possible forms are 'cylindrical' ('h' and 'orient' needed), ...
- **h** – needed for a cylindrical cluster shape. Specifies the height of the cylinder. Default=0. Input in units of the lattice constant.
- **vector** – needed for a cylindrical cluster shape. Specifies the orientation vector of the cylinder. Default: z-direction.

Returns array with all the atoms within the cutoff (x_res)

ToDo

- dynamical box construction (r_cut determines which values n1, n2, n3 have)
- different cluster forms (spherical, cylinder, ...), add default parameters, better solution for 'orient'

aiida_kkr.tools.tools_kkrimp.**get_distance**(*structure_array*, *i*, *j*)

Calculates and returns the distances between to atoms i and j in the given structure_array

Parameters **structure_array** – input numpy array of the cell containing all the atoms ((# of atoms) x 6-matrix)

Params i, j indices of the atoms for which the distance should be calculated (indices again start at 0)

Returns distance between atoms i and j in units of alat

Note

aiida_kkr.tools.tools_kkrimp.**get_structure_data**(*structure*)

Function to take data from AiiDA's StructureData type and store it into a single numpy array of the following form: a = [[x-Position 1st atom, y-Position 1st atom, z-Position 1st atom, index 1st atom, charge 1st atom, 0.],

[x-Position 2nd atom, y-Position 2nd atom, z-Position 2nd atom, index 2nd atom, charge 1st atom, 0.], [..., ..., ..., ..., ..., ...], ...]

Parameters **structure** – input structure of the type StructureData

Returns numpy array a[# of atoms in the unit cell][5] containing the structure related data (positions in units of the unit cell length)

Note

aiida_kkr.tools.tools_kkrimp.**make_scoef**(*structure, radius, path, h=-1.0, vector=[0.0, 0.0, 1.0], i=0, alat_input=None*)

Creates the 'scoef' file for a certain structure. Needed to conduct an impurity KKR calculation.

Parameters

- **structure** – input structure of the StructureData type.
- **radius** – input cutoff radius in Ang. units.
- **h** – height of the cutoff cylinder (negative for spherical cluster shape). For negative values, clust_shape will be automatically assumed as 'spherical'. If there will be given a h > 0, the clust_shape will be 'cylindrical'.
- **vector** – orientation vector of the cylinder (just for clust_shape='cylindrical').
- **i** – atom index around which the cluster should be centered. Default: 0 (first atom in the structure).
- **alat_input** – input lattice constant in Ang. If *None* use the lattice constant that is automatically found. Otherwise rescale everything.

class aiida_kkr.tools.tools_kkrimp.**modify_potential**

Class for old modify potential script, ported from modify_potential script, initially by D. Bauer

__weakref__

list of weak references to the object (if defined)

neworder_potential(*potfile_in*, *potfile_out*, *neworder*, *potfile_2=None*, *replace_from_pot2=None*, *debug=False*)

Read potential file and new potential using a list describing the order of the new potential. If a second potential is given as input together with an index list, then the corresponding of the output potential are overwritten with positions from the second input potential.

Parameters

- **potfile_in** (*str*) – absolute path to input potential
- **potfile_out** (*str*) – absolute path to output potential
- **neworder** (*list*) – list after which output potential is constructed from input potential
- **potfile_2** (*str*) – optional, absolute path to second potential file if positions in new list of potentials shall be replaced by positions of second potential, requires *replace_from_pot* to be given as well
- **replace_from_pot** (*list*) – optional, list containing tuples of (position in newlist that is to be replaced, position in pot2 with which position is replaced)

Usage

1. modify_potential().neworder_potential(<path_to_input_pot>, <path_to_output_pot>, [])

shapefun_from_scoef(*scoefpath*, *shapefun_path*, *atom2shapes*, *shapefun_new*)
Read shapefun and create impurity shapefun using scoef info and shapes array

Parameters

- **scoefpath** – absolute path to scoef file
- **shapefun_path** – absolute path to input shapefun file
- **shapes** – shapes array for mapping between atom index and shapefunction index
- **shapefun_new** – absolute path to output shapefun file to which the new shapefunction will be written

aiida_kkr.tools.tools_kkrimp.**rotate_onto_z**(*structure*, *structure_array*, *vector*)
Rotates all positions of a structure array of orientation 'orient' onto the z-axis. Needed to implement the cylindrical cutoff shape.

Parameters

- **structure** – input structure of the type StructureData
- **structure_array** – input structure array, obtained by select_reference for the referenced system.
- **vector** – reference vector that has to be mapped onto the z-axis.

Returns rotated system, now the 'orient'-axis is aligned with the z-axis

aiida_kkr.tools.tools_kkrimp.**select_reference**(*structure_array*, *i*)
Function that references all of the atoms in the cell to one particular atom i in the cell and calculates the distance from the different atoms to atom i. New numpy array will have the form: x = [[x-Position 1st atom, y-Position 1st atom, z-Position 1st atom, index 1st atom, charge 1st atom,

distance 1st atom to atom i],

[x-Position 2nd atom, y-Position 2nd atom, z-Position 2nd atom, index 2nd atom, charge 1st atom,
distance 1st atom to atom i],

[..., ..., ..., ..., ..., ...], ...]

Parameters

- **structure_array** – input array of the cell containing all the atoms (obtained from get_structure_data)
- **i** – index of the atom which should be the new reference

Returns new structure array with the origin at the selected atom i (for KKRimp: impurity atom)

Note the first atom in the structure_array is labelled with 0, the second with 1, ...

aiida_kkr.tools.tools_kkrimp.**write_scoef**(*x_res*, *path*)
Sorts the data from find_neighbors with respect to the distance to the selected atom and writes the data correctly formatted into the file 'scoef'. Additionally the total number of atoms in the list is written out in the first line of the file.

Parameters **x_res** – array of atoms within the cutoff radius obtained by find_neighbors (unsorted)

Output returns scoef file with the total number of atoms in the first line, then with the formatted positions, indices, charges and distances in the subsequent lines.

aiida_kkr.tools.tools_kkrimp.**write_scoef_full_imp_cls**(*imp_info_node*, *path*, *rescale_alat=None*)
write scoef file from imp_cls info in imp_info_node

Plotting tools

contains plot_kkr class for node visualization

aiida_kkr.tools.plot_kkr.**_check_tk_gui**(*static*)
check if tk gui can be openen, otherwise reset static to False this is only needed if we are not inside a notebook

aiida_kkr.tools.plot_kkr.**_has_ase_notebook**()
Helper function to check if ase_notebook is installed

aiida_kkr.tools.plot_kkr.**_in_notebook**()
Helper function to check if code is executed from within a jupyter notebook this is used to change to a different default visualization

aiida_kkr.tools.plot_kkr.**plot_imp_cluster**(*kkrimp_calc_node*, ***kwargs*)
Plot impurity cluster from KkrimpCalculation node

These kwargs can be used to control the behavior of the plotting tool:

kwargs = { static = False, # make gui or static (svg) images canvas_size = (300, 300), # size of the canvas zoom = 1.0, # zoom, set to >1 (<1) to zoom in (out) atom_opacity = 0.95, # set opacity level of the atoms, useful for overlapping atoms rotations = "-80x,-20y,-5z", # rotation in degrees around x,y,z axes show_unit_cell = True, # show the unit cell of the host filename = 'plot_kkr_out_impstruc.svg' # filename used for the export of a static svg image

}

class aiida_kkr.tools.plot_kkr.**plot_kkr**(*nodes=None*, ***kwargs*)
Class grouping all functionality to plot typical nodes (calculations, workflows, ...) of the aiida-kkr plugin.

Parameters **nodes** – node identifier which is to be visualized

optional arguments:

Parameters

- **silent** (*bool*) – print information about input node including inputs and outputs (default: False)
- **strucplot** (*bool*) – plot structure using ase's view function (default: False)
- **interpol** (*bool*) – use interpolated data for DOS plots (default: True)
- **all_atoms** (*bool*) – plot all atoms in DOS plots (default: False, i.e. plot total DOS only)
- **l_channels** (*bool*) – plot l-channels in addition to total DOS (default: True, i.e. plot all l-channels)
- **sum_spins** (*bool*) – sum up both spin channels or plot both? (default: False, i.e. plot both spin channels)
- **logscale** – plot rms and charge neutrality curves on a log-scale (default: True)
- **switch_xy** (*bool*) – (default: False)
- **iatom** (*list*) – list of atom indices which are supposed to be plotted (default: [], i.e. show all atoms)

additional keyword arguments are passed onto the plotting function which allows, for example, to change the markers used in a DOS plot to crosses via *marker='x'*

Usage plot_kkr(nodes, **kwargs)

where nodes is a node identifier (the node itself, it's pk or uuid) or a list of node identifiers.

Note If nodes is a list of nodes then the plots are grouped together if possible.

__init__ (*nodes=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

classify_and_plot_node (*node*, *return_name_only=False*, ***kwargs*)
Find class of the node and call plotting function.

dosplot (*d*, *natoms*, *nofig*, *all_atoms*, *l_channels*, *sum_spins*, *switch_xy*, *switch_sign_spin2*, ***kwargs*)
plot dos from xydata node

get_node (*node*)
Get node from pk or uuid

get_rms_kkrcalc (*node*, *title=None*)
extract rms etc from kkr Calculation. Works for both finished and still running Calculations.

group_nodes (*nodes*)
Go through list of nodes and group them together.

make_kkrimp_rmsplot (*rms_all*, *stot_all*, *pks_all*, *rms_goal*, *ptitle*, ***kwargs*)
plot rms and total spin moment of kkrimp calculation or series of kkrimp calculations

plot_group (*groupname*, *nodesgroups*, ***kwargs*)
Visualize all nodes of one group.

plot_kkr_calc (*node*, ***kwargs*)
plot things for a kkr Calculation node

plot_kkr_dos (*node*, ***kwargs*)
plot outputs of a kkr_dos_wc workflow

plot_kkr_eos (*node*, ***kwargs*)
plot outputs of a kkr_eos workflow

plot_kkr_scf (*node*, ***kwargs*)
plot outputs of a kkr_scf_wc workflow

plot_kkr_single_node (*node*, ***kwargs*)
TODO docstring

plot_kkr_startpot (*node*, ***kwargs*)
plot output of kkr_startpot_wc workflow

plot_kkrimp_calc (*node*, *return_rms=False*, *return_stot=False*, *plot_rms=True*, ***kwargs*)
plot things from a kkrimp Calculation node

plot_kkrimp_dos_wc (*node*, ***kwargs*)
plot things from a kkrimp_dos workflow node

plot_kkrimp_sub_wc (*node*, ***kwargs*)
plot things from a kkrimp_sub_wc workflow

plot_kkrimp_wc (*node*, ***kwargs*)
plot things from a kkrimp_wc workflow

plot_struc (*node*, ***kwargs*)
visualize structure using ase's *view* function

plot_voro_calc(*node*, ***kwargs*)
plot things for a voro Calculation node

print_clean_inouts(*node*)
print inputs and outputs of nodes without showing 'CALL' and 'CREATE' links in workflows.

rmsplot(*rms*, *neutr*, *nofig*, *ptitle*, *logscale*, *only=None*, *rename_second=None*, ***kwargs*)
plot rms and charge neutrality

aiida_kkr.tools.plot_kkr.**save_fig_to_file**(*kwargs*, *filename0='plot_kkr_out.png'*)
save the figure as a png file look for filename and static in kwargs save only if static is True after _check_tk_gui check to make it work in the command line script

aiida_kkr.tools.plot_kkr.**strucplot_ase_notebook**(*struc*, ***kwargs*)
plotting function for aiida structure using ase_notebook visulaization

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

E

F

G

H

I

K

M

N

P

R

S

T

U

V

W