DMFT-Wien2k Manual

Kristjan Haule
Department of Physics, Rutgers University, Piscataway, NJ 08854, USA

(Dated: December 26, 2009)

PACS numbers: 71.27.+a,71.30.+h

I. INSTALATION

The program consists of many independent programs, which are written in C++, fortran90, and Python.

Here are some important steps to instal the package

- Edit the file `Makefile.in` to set the path to compilers, compiler options, and libraries on your system. You will need
  - intel mkl library
  - intel fortran compiler
  - gnu C++ compiler
  - Python with numpy and scipy
  - Python CXX package
- type `make`
- set an environment variable `WIEN_DMFT_ROOT` to point to the directory you plan to install the code
- type `make install`

Note: When installing numpy on a linux distribution, make sure to have gcc in the environment variable CC=gcc and CXX=g++, because otherwise linux will install f2py compiled with icc and icpc, which does NOT work at all (the options sent to icc are wrong at compilation time of f2py).

II. SHORT INTRO TO RUNNING DMFT-WIEN2K CODE

The package consists of many independent modules, which are written in C++, fortran90, and Python.

The highest level scripts are written in Python (".py" files). You can always get help on python script by running the script with the argument "-h" or "−−help".

The most important steps of the LDA+DMFT executions are:

- **initialization**: After LDA is converged with the Wien2K package, one should run
  `> init_dmft.py`
  and follow the instructions. The script will create two input files: *case*.indmfl and *case*.indmfi . The first contains information of how a local self-energy can be added to the Kohn-Sham potential. The second connects the local self-energy with the output self-energy of the impurity solvers.
- **preparation**: Prepare additional input files
  - `params.dat`
    The file must contains information for the impurity solver, number of self-consistent steps, etc.
  - `sig.inp`
    Contains starting guess for the input self-energy. Here zero dynamic self-energy is usualy a good starting point. This can be generated by
    `> szero.py` -e Edc
    "Edc" should be a number close to $U(n - 1/2)$, where n is expected impurity occupation. This creates a good guess for double-couting and $\Sigma(\infty)$, but ultimately a good guess for the impurity levels.

    The frequency mesh for the self-energy is very important. It is generated in the following way:
    If you have a good self-energy from some other run, you can copy it to the working directory, and the script will take the mesh from current *sig.inp*. If *sig.inp* does not exist, you should specify the following arguments to the *szero.py* script
    - -n int : Number of frequency points
    - -T float : Temperature for the imaginary axis mesh
    - -L float : cuttof energy on the real axis

    The real-axis mesh generated in this way is not efficient, and one should rather generatea more efficient "tan" mesh with alternative script, and copy it to *sig.inp.*
  - `Sigma.000`
    This file is needed only on real axis. It is a guess for the self-energy of pseudoparticles. If you have no experience in generating this file, you should take it from some example run. You only need the first column, which gives a frequency for the self-energy.
- **self-consistent run**: By invoking
  `> run_dmft.py`
  the python script will produce self-consistent LDA+DMFT solution.

  Useful informtation is stored in the following log files:

`dmft_info.out` – top-most information about the LDA+DMFT run
case.`dayfile` – list of all executed steps and current convergence.
`dmft1_info.out` – information about the dmft1 step
case.`outputdmf1` – more information on dmft1 step from fortran routines.
`dmft2_info.out` – information about the dmft2 step
case.`outputdmf2` – more information on dmft2 step from fortram routines.

The self-consistent calculation, performed by (`run_dmft.py`) can also be performed by a few steps, which can be invoked sequentially by the user. These steps are

- *LDA potential* : Is computed by
  `> x lapw0`
- *LDA eigensystem* : Is computed by
  `> x lapw1`
- *so-coupling* : When needed, so-coupling ss added by
  `> x lapwso`
- self-energy split: Is invoked by
  `> ssplit.py`

  The self-enery for all atoms and all orbitals is stored in `sig.inp`. The first two lines contain the double-couting E_{dc}, and $\Sigma(\infty)$. The columns correspond to the dynamic part of the self-energy. Each correlated block $(atom, l)$ needs an independent input file in the `dmft1` and `dmft2` step. Even if two atoms are equivalent, they need independent input self-energy. For each such block, a file `sig.inp[r]` is generated (where `[r]`is positive integer), which contains $\Sigma(\omega) - E_{dc}$ for each correlated block.
- **dmft1 step** : Can be invoked by
  `> x_dmft.py dmft1`

  Computes the local Green's function and the hybridization function. The ouput files are
  case.`cdos` – density of states,
  case.`gc[r]` – local green's functios,
  case.`dlt[r]` – hybridization function,
  case.`Eimp[r]` – impurity levels,
  case.`outputdmf1` – logging information,
- Prepare impurity hybridization: Invoked by
  `> sjoin.py -m` mixing-parameter
  It produces hybridization function for all impurity problems. `dmft1` step produces hybridization function and impurity levels for all correlated blocks (*case*..`dlt[r]`). In DMFT, the number of impurity problems can be smaller than the number of correlated atoms (either because some atoms are equivalentm, or some atoms are grouped together in clusters). From the hybridization functions (*case*..`dlt[r]`, *case*..`Eimp[r]`), the impurity hybridization (`imp.[r]/Delta.imp`, `imp.[r]/Eimp.inp`) are generated.
- ***Impurity solver*** : Solves the auxiliary impurity problem. Currently supported impurity solvers are:
  - CTQMC – continuous time quantum Monte Carlo
  - OCA – the one crossing approximation
  - NCA – To invoke it, the mode should be "OCA", but the executable should be "nca".
- combine impurity self-energies : Invoked by
  `> sgather.py`
  It take the impurity self-energy, and creates a common file with self-energy. The impurity solvers produce the new self-energy in `imp.[r]/sig.out`. The result is combined into a single file named `sig.inp` .
- self-energy split: Invoked by
  `> ssplit.py`
  In the next step (dmft2) we want to use the new self-energy just produced by the impurity solver. Hence, we create `sig.inp[r]` again from single self-energy file `sig.inp`, just created.
- **dmft2 step** : Recomputes the electronic charge using LDA+DMFT self-energy. The output is stored in
  case.`clmval` – the new valence charge density
  `EF.dat` – the new chemical potential
  case.`cdos3` – occupied part of the DOS (just for debuging purposes.)
  `dmft2_info.out` – some logging information
  case `.outputdmf2` – more logging information from the fortram subroutines.
  case `scf2` – more information from the fortran.
- **lcore** : Computes LDA core density by invoking
  `> x lcore`
- **mixer** : Produces the total electronic charge, and mixes it with previous density by broyden-like method. Invoked by:
  `> x mixer`