QuantumEspressoTools.jl v0.2 – A Julia package for automation of QuantumEspresso calculations

Yunlong Lian
September 2024

Abstract

`QuantumEspressoTools.jl` is a Julia package for automation of `Quantum Espresso` (`QE`) calculation tasks. The main purpose is to facilitate the launching of `QE` task sequence on nodes of a high-performance computing cluster to perform particular types of calculations, such as structural relaxation, electron and phonon band structures under similar settings. `QuantumEspressoTools.jl` provides many useful functions to connect different `QE` calculators and allows the user to build up comstomized task sequence. The present version (v0.2) is capable of performing energy (enthalpy) calculations under different settings (k-mesh, pressure, etc.), structural relaxations with gradually tightening precesion controls and structural perturbations, electron band structure calculation and analysis (component, DoS, Wannier, etc.), as well as phonon frequencies. Other types of calculation, such as *abinitio* molecular dynamics and electron-phonon coupling, will be included in future updates.

Contents

Chapter 1

Introduction

1.1 A survey on automated abinitio calculations

In recent years, the speed of *abinitio* calculations has increased drastically, making high-throughput calculations on material classes possible. This type of calculation relies on automation of elementary tasks using the *abinitio* calculation softwares, such as `VASP`, `Quantum Espresso` and `Abinit`. Several tools for automated *abinitio* calculation are available, namely `ASE` and `AiiDA`. They provide many functions and can be easily used to build up high-throughput calculation schemes. In this section, I give a brief survey on these libraries.

`ASE`, or `Atomic Simulation Environment`, is an open-source Python software package to facilitate atomistic simulations in all stages. It provides high-level abstractions for atoms and *abinitio* calculators, which seal the technical details and help the user to focus on the physical aspects of the calculations. One can use `ase.Atoms` and routines in `ase.build` to generate crystals, surfaces and interfaces, or use functions in `ase.db` to retrieve databases of materials. One can also use calculators in `ase.calculators` to compute energy and force of a structure, or perform optimizations or structural dynamics simulations under various temperature and pressure settings. These features covers a wide range of applications of abinitio calculations.

`AiiDA` is "an open-source, high-throughput, scalable computational infrastructure for automated reproducible (*abinitio* calculation) workflows and data provenance".[1] It is written in [...]

[1] Martin Uhrin, et. al., "Workflows in AiiDA: Engineering a high-throughput, event-based engine for robust and modular computational workflows", Computational Materials Science Volume 187, 1 February 2021, 110086. DOI: 10.1016/j.commatsci.2020.110086

QuantumEspressoTools is an open-source Julia package for automation of Quantum Espresso (QE) calculation tasks. It aims at orchestrating QE calculations on high-performance computing clusters. The present version is an assembly of Julia codes used in my research projects. Modest abstractions are made on the representation of input configuration scripts and the task sequence. Much efforts are made to reliably extract the quantitative results from the output of QE programs.

1.2 Examples calculations with Quantum Espresso

In a typical *abinitio* calculation, one has to prepare the input script and analyze the output files. These procedures are usually done manually, typically by modifying a seed file to prepare for input and running various scripts to read the output, resulting in various errors and waste of CPU hours on the supercomputer. To reveal the tedious manual procedures, I provide two case studies of QE calculations, which had eventually motivated the earliest version of QuantumEspressoTools.

The first example is the structural relaxation. A common wisdom to use pw.x for structural relaxation is that a single program run is insufficient to determine the optimal structure — one often has to repeat the calculation until the final structure does not change. Since the repeated runs of pw.x cannot always be scheduled without gaps, they require many interventions by the user and delay the overall progress.

The second example is the component analysis of electron bands. In the analysis, the key specifications, namely the atomic orbits and energy windows, must be mannually determined based on the previous results. The situation is similar as in the previous case.

1.3 Highlights of QuantumEspressoTools and plan of development

In QuantumEspressoTools the following QE calculation workflows have been fully implemented:

- pseudopotential, k-point and energy cutoff test (pw.x)
- energy (enthalpy) calculation of similar structures with change of pressure (pw.x)
- structural relaxation with gradually tightening precesion controls and structural perturbations (pw.x)

- electron band structure calculation and analysis (`pw.x`, `projwfc.x`, `pw2wannier90.x` and `wannier90.x`)
- phonon frequency calculations and imaginary frequency detection (`ph.x`)
- pseudopotential generation and test (`ld1.x` and `pw.x`)

Based on these implementations, stable computation services are currently running on a private supercomputing cluster in CSNS.

The automation is based on a simple sequential computation model. In this model, an *abinitio* calculation task is represented by one or several *instructions*. The entire calculation is therefore represented by a *program*. Essentially, the automation of *abinitio* calculation is to design a *language* $L = (S, G)$ that allows the user to write *programs* according to the grammar G using a set of *instructions* S.[2] The current version v0.2 of `QuantumEspressoTools` has an elementary implementation of the language. Future developments will be focused on the mapping between the (composite) *abinitio* calculation and the programming language on the level of instruction and grammar.

Recently, the machine learning applications of automated abinitio calculation has been in high demand. In particular, considerable amount of abinitio calculations should be performed in a controlled way to generate training sets for the neural-network representation of interatomic potentials. The bottleneck is the cost in the *abinitio* calculation part. Future development of `QuantumEspressoTools` will be guided by machine learning applications.

On the technical level, several improvements are scheduled. The Fortran-Julia data interface will be separated from the package to insulate the I/O bugs from propagating to the core modules of `QuantumEspressoTools`. The event log module will be completely redesigned such that the entire calculation can be reconstructed from the text log file. Inter-node communication facilities on HPC will be integrated to the package to enable `QuantumEspressoTools` to orchestrate calculations in larger scale.

[2] Theory of Computation.

Chapter 2

Design and implementations of `QuantumEspressoTools`

This chapter is a documentation of important design ideas and implementation details of `QuantumEspressoTools`.

2.1 Principles and overview of the design

Top-level design goals. An elementary *abinitio* calculation task can be, at highest level of abstraction, formulated as an *instruction*. For example, the following instruction calls `pw.x` to perform self-consistent calculation on structure `strct` with settings `sttng` and assigns the result to `res`:

```
# perform self-consistent calculation on structure `strct`
# with settings `sttng`
# to get result `res`
res := SCF (strct) WITH (sttng)
```

The second example is the following `RELAX` instruction, which is trying to relax the structure `strct_0` under constraint `cnstr` to get `strct_1`:

```
# relax the structure `strct_0` to get `strct_1`
# under the constraint `cnstr`
strct_1 := RELAX (strct_0) UNDER (cnstr)
```

The third example could be the operations on structures:

```
# interpolating two structures strct_0 and strct_1
# (with same chemical formula)
t := 0.3
strct_interpol := t * (strct_0) + (1-t) * (strct_1)

# creates a virtual crystal SixGe1-x
# for calculations with the virtual crystal approximation
x := 0.5
strct_VCA := x * (strct_Si) <> (1-x) * (strct_Ge)

# generate a random structure `d_strct`
# according to `strct_0` and modify it
d_strct := RANDOM (strct_0)
strct_1 := (strct_0) << 0.1 * (d_strct)

# distort the structure `strct`
# by applying strain according to `strain_tensor`
strct_1 := STRAIN (strct_0) BY (strain_tensor)
```

where the interpolation, VCA superposition and update operations on structures are denoted as `+`, `<>` and `<<`, respectively.

By treating calculations as black boxes and formulating them as instructions, `QuantumEspressoTools` allows the user to write high-level programs for a complicated calculation. For example, Alg.2.1 searches for the optimal path in structural transition between `A` and `B`. Another example would be Alg.2.2, which describes the finite displacement method[1] to compute the force constants between pairs of atoms in a crystal. Yet another example is Alg.2.3 for the method of quasi-random structures.[2] The final example is Alg.2.4, which is the self-consistent calculation of the Hubbard-U parameter.[3] Compared to the common practice of organizing calculations by bash scripts, the instruction-based formulation is self-explanatory and more robust. `QuantumEspressoTools` uses instructions to model elementary `QE` calculations, in order to minimize the efforts for automation and maximize the range of its applications.

[1] https://phonopy.github.io/phonopy/formulation.html#modified-parlinski-li-kawazoe-method

[2] Michael C. Gao, Changning Niu, Chao Jiang, Douglas L. Irving, Applications of Special Quasi-random Structures to High-Entropy Alloys, in *High-Entropy Alloys*, Springer 2016. DOI https://doi.org/10.1007/978-3-319-27013-5

[3] http://hjkgrp.mit.edu/content/calculating-hubbard-u

Algorithm 2.1 Optimal path for structural transition between `A` and `B`, with maximal search depth `d`.

```
# optimal path in structural phase transition
# between two structures `A` and `B`
function PATH(A::Structure, B::Structure, d::Int)
    ENERGY(X) := SCF (X) WITH (global_setting)
    PERTURB(X, Y) := generating structure perturbation of X according to Y
    if d==0 # linear interpolation
        ### ABINITO ###
        topt := argmax(ENERGY(t*A+(1-t)*B) for t in [0,1])
        Sopt := topt*A+(1-topt)*B
        return [A,Sopt,B], ENERGY(Sopt)
        ###############
    else # recursive
        if d%2==0
            dBopt := argmin(PATH(A, B+dB, d-1) for dB in PERTURB(B, A))
            S1opts, EN1 := PATH(A, B+dBopt, d-1)
            S2opt,  EN2 := PATH(B+dBopt, B, 0)
            return [A, S1opts..., B+dBopt, S2opt, B], min(EN1,EN2)
        else
            dAopt := argmin(PATH(A+dA, B, d-1) for dA in PERTURB(A, B))
            S1opts, EN1 := PATH(A+dAopt, B, d-1)
            S2opt,  EN2 := PATH(A, A+dAopt, 0)
            return [A, S2opt, A+dAopt, S1opts..., B], min(EN1,EN2)
        end
    end
end
```

Algorithm 2.2 Finite displacement method to calculate the force constants between pairs of atoms in a crystal.

```
# finite displacement method
function FC(
    strct0::Structure,                  # initial structure
    spcell_dim::Tuple{Int,Int,Int},     # supercell dimensions
    sttng::Config,                      # scf calcl settings
    disp::Real                          # dinite displacement amplitude
    )
    # supercell
    strct := ENLARGE (strct0) BY (spcell_dim)
    # initial force (not necessarily zero)
    force0 := SCF (strct) WITH (sttng)
    # compute forces for all finite displacements
    diff_fc := []
    for atom in strct
        for direction in [x,y,z]
            # update `strct`
            strct_disp = strct << disp * MOVE(strct, atom, direction)
            ### ABINITO ###
            force := SCF (strct_disp) WITH (sttng)
            ###############
            push!(diff_fc, (atom, direction, (force-force0))
        end
    end
    # compute the pairwise forces
    # [ref] https://phonopy.github.io/phonopy/formulation.html
    force_constant := (1/disp) * EXTRACT_PAIRWISE_FORCE(diff_fc)
    return force_constant
end
```

Algorithm 2.3 Method of quasi-random structures used in the study of high-entropy alloys.

```
# method of quasi-random structures
function QS(
    strct::Structure,               # seed structure
    molratio::Dict{Element,Real},   # element => molar ratio
    common_setting::Config,         # scf calcl settings
    N_samples = 1000,               # number of samples
    is_relax = true                 # relax the updated structure?
    )
    ENERGY(X) := SCF (X) WITH (common_setting)
    samples := []
    for i = 1:N_samples
        strct := RANDOM_UPDATE(strct, molratio)
        ### ABINITO ###
        if is_relax
            strct := RELAX (strct) UNDER (common_setting)
        end
        push!(samples, (strct, ENERGY(strct)))
        ###############
    end
    return STATISTICAL_PROPERTY(samples)
end
```

Algorithm 2.4 Self-consistent determination of Hubbard-U parameter.

```
# http://hjkgrp.mit.edu/content/calculating-hubbard-u
# linear-response calculation of Hubbard-U
# M. Cococcioni, S. de Gironcoli. PRB.71,035105(2005)
function Hubbard_U_linear_response(
    strct::Structure,
    sttng::Config
    )
    n0_list = []
    n_list  = []
    for a in alpha_list
        sttng := UPDATE (sttng) BY (alpha = a)
        sttng := UPDATE (sttng) BY (U = 0)
        push!(n0_list, SCF (strct) WITH (sttng) GET FIRST ("nsum"))
        push!(n_list,  SCF (strct) WITH (sttng) GET LAST  ("nsum"))
    end
    # chi = d n / d alpha
    chi0 = LIN_FIT_SLOPE(n0_list, alpha_list)
    chi  = LIN_FIT_SLOPE(n_list,  alpha_list)
    # U = 1/chi0 - 1/chi
    U_linear_response = 1.0/chi0 - 1.0/chi
    return U_linear_response
end

# self-consistent calculation of Hubbard-U
# H. J. Kulik, et. al., PRL.97,103001(2006)
function Hubbard_U_self_consistent(
    strct::Structure,
    sttng::Config,
    U_list::Vector{Float64}
    )
    U_out_list  = []
    for U in U_list
        sttng := UPDATE (sttng) BY (alpha = 0)
        sttng := UPDATE (sttng) BY (U = U)
        push!(U_out_list, SCF (strct) WITH (sttng) GET ("U"))
    end
    U_self_consistent = LIN_FIT_INTERCEPT(U_out_list, U_list)
    return U_self_consistent
end
```

Design in the present version v0.2. The instruction-based design discussed above requires a *language interpreter* and is not easy to implement. Due to limited time on source code development and urgent demand in research projects, I used an intermediate-level abstraction for the elementary QE calculations. All the composite QE calculations are considered as a sequence of elementary tasks, without branching or loop. The task sequence is then represented by the data structure INSTRUCTIONS listed in Alg.**??**, which is a *static* set of instructions.[4] The serial execution of instructions is illustrated in Fig.2.1.

The source code of the QuantumEspressoTools package is delivered in two parts, namely the ./src folder and the ./scripts folder. In the ./src folder, input script generators and output file analyzers are written case by case for each QE programs (pw.x, ph.x, cp.x, etc.). There are also codes to querry the pseudopotential files and to deal with crystallographic symmetries in this folder. The functions for various types of calculations have been written as external scripts and they are collected in the ./scripts folder. The final design may require significant modifications on these functions. Fig.2.2 summarizes the architecture of the package.

[4] In future versions, the instruction will be made *dynamic* to mimic a Turing machine.

Algorithm 2.5 Data structure `INSTRUCTION`.

```
INSTRKEY = Union{Symbol,String}
mutable struct INSTRUCTION
    # (system command to execute QE program) =>
    # (default configurations)
    SEED::Dict{Cmd, Dict{INSTRKEY,Any}}
    # [(title1, command1, spec_config1), (title2, command2, spec_config2), ...]
    MODIFY::Vector{Tuple}
end
```

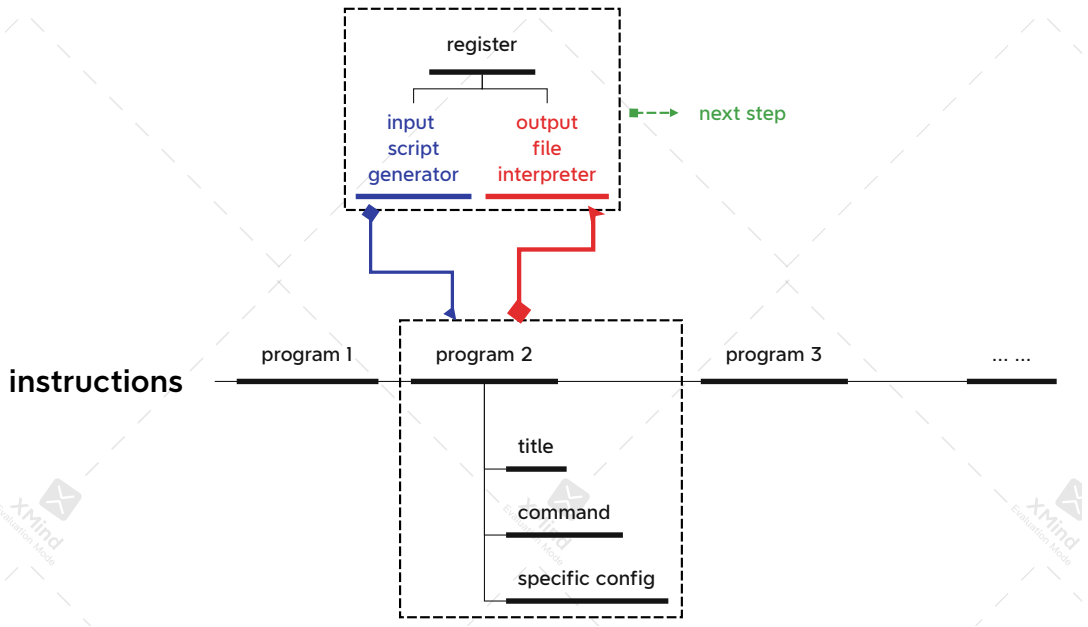

Figure 2.1: Serial execution of instructions.

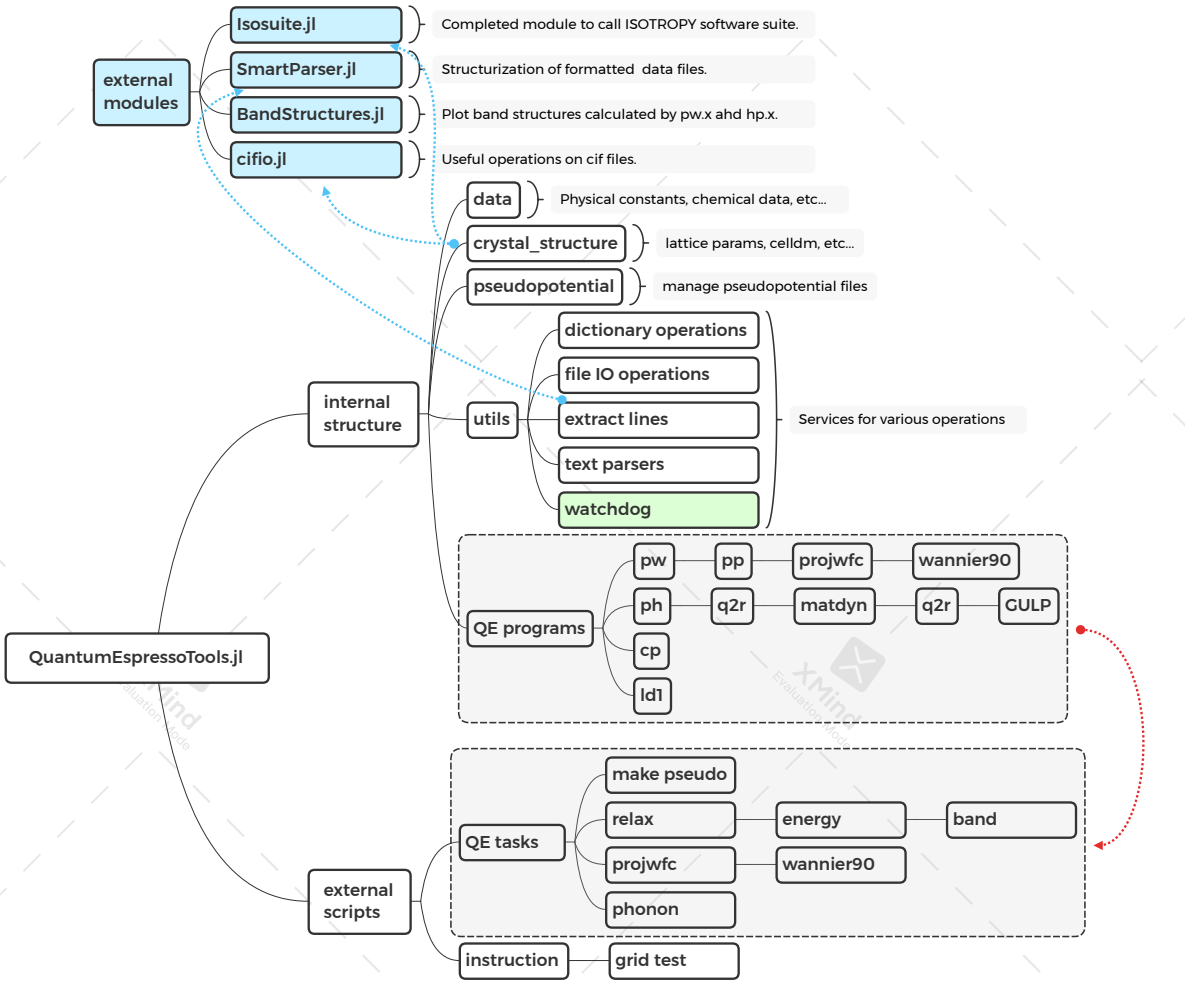

Figure 2.2: Architecture of `QuantumEspressoTools v0.2`.

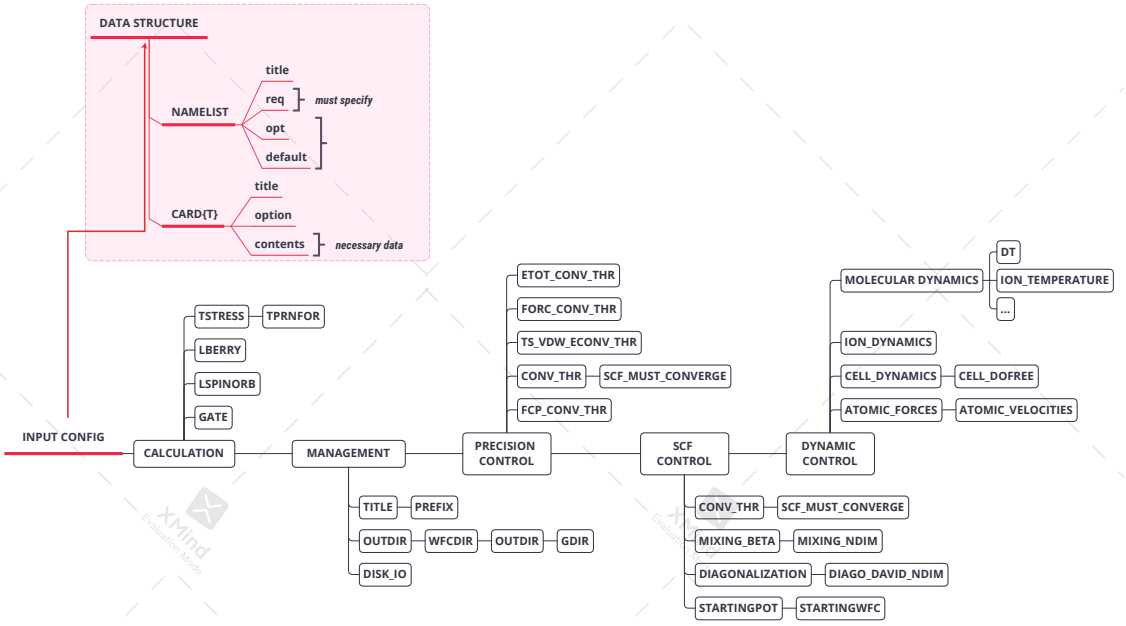

Figure 2.3: Overview of input options for `pw.x`. (part 1)

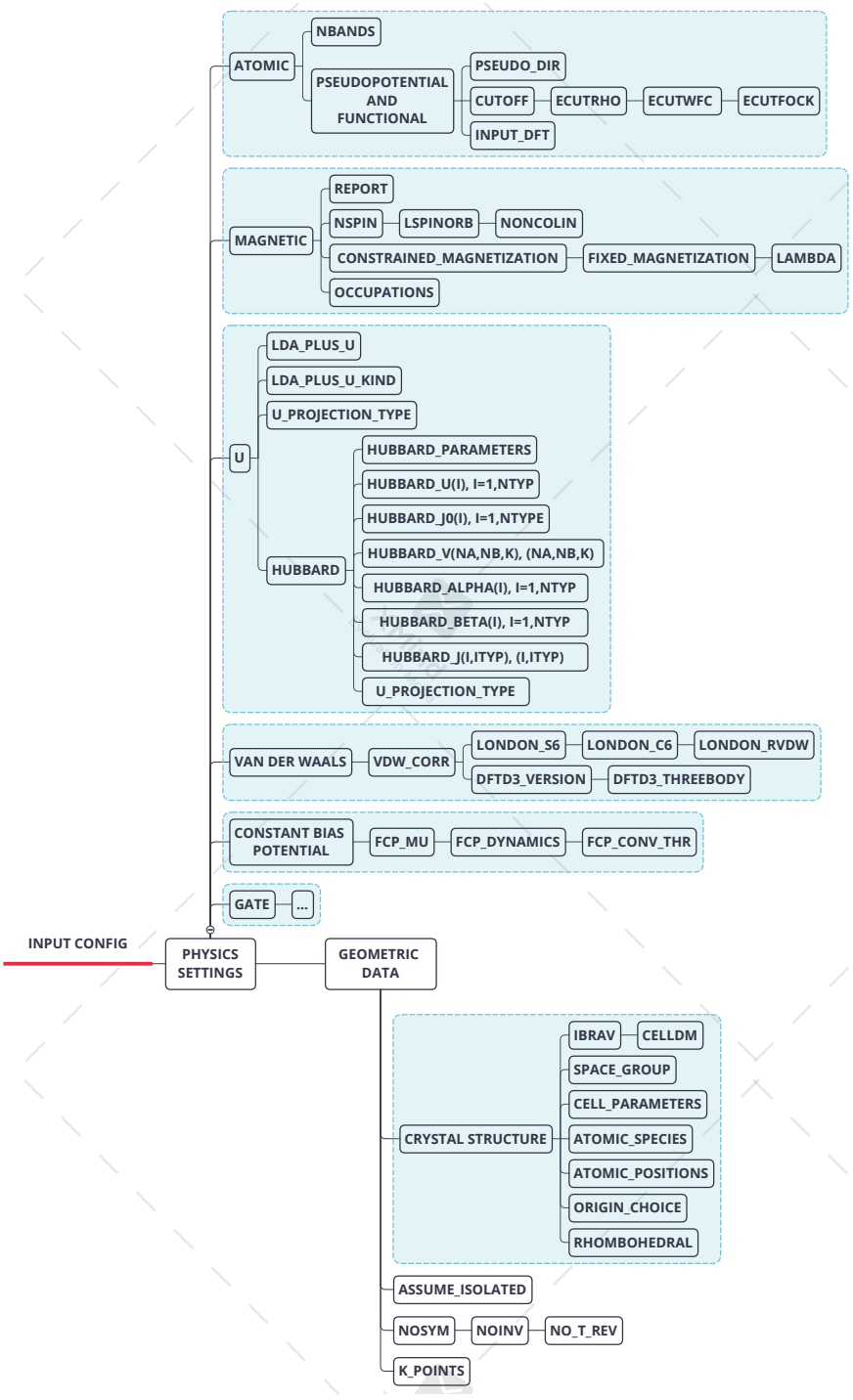

Figure 2.4: Overview of input options for `pw.x`. (part 2)

2.2 Implementation details

2.2.1 Elementary QE calculation

Preparation of pw.x input script The options of pw.x is summarized in Fig.2.3 and Fig.2.4. The main difficulty of configuring pw.x is the interdependency of these options. In QuantumEspressoTools, considerable effort has been made to generate conflict-free input scripts from user specifications. In particular, the options related to pseudopotentials are simplified, such that the user only have to provide a label to specify the type of pseodupotential and functional. The ibrav option is related to the space group of the input structure, since the later is used more frequently.

Analyze the pw.x output The automation of QE calculations relies on the analysis of output files, since the input script for next calculation depends on the information extracted from the present output files. In manually maintained project with QE, considerable time is spent on connecting elementary calculations and numerous errors are often introduced. In QuantumEspressoTools, a set of functions are carefully written to extract particular pieces of information from the pw.x output file. Each type of calculation often has definite prerequisite information. Provided reliable analyzers of output files, the elementary QE calculations can be automated.

Other QE programs The automations for input script preparation and output file analysis have to be done case by case, following the example of pw.x.

2.2.2 Watchdog

The watchdog is a monitor on the output file of a running QE program. It checks the specified file at a given rate within the maximal time. It terminates the program if the user-specified termination condition is satisfied. When returns, it reports the status of the program being watched. The watchdog is crucial to the robustness of the automation.

The watchdog only monitors the output file of the QE program and does not communicate with any running processes, unless when the program must be terminated under certain conditions. This is a platform-independent way of monitoring QE programs. It is the most robust design of a watchdog.

2.2.3 Instructions and execution

The current implementation of instructions is, as mentioned earlier, is a static collection. The execution of instruction `S::INSTRUCTION` of such type is described in Alg.2.6.

Algorithm 2.6 Serial execution of `S::INSTRUCTION`.

```
expand(S::INSTRUCTION) = [
    (title,program,update(S.SEED[program],config))
    for (title,program,config) in S.MODIFY ]

function execute_serial(
    S::INSTRUCTION;
    WORKSPACE=".",
    from_scratch=false
    )
    RESULTS   = []
    REGISTER  = Dict()
    for (title, program, config) in expand(S)
        # variables
        folder = create_folder_in_workspace(WORKSPACE, title)
        configuration = Dict("outdir"=>folder) << config << REGISTER
        updater = configuration[:updater]
        # prepare and launch QE program
        input_scripts = generate_input_script(program, configuration)
        output_lines, success, status =
            lauch_program(program, input_scripts, from_scratch)
        results  = interprete_results(program, output_lines)
        REGISTER = updater(REGISTER, output_lines)
        # record results
        push!( RESULTS,
               (title, program, config,
                input_scripts, output_lines, REGISTER, results) )
        # exit if not success or instructed
        !(success) && break
        (:exit in keys(REGISTER)) && REGISTER[:exit] && break
    end
    return RESULTS
end
```

Chapter 3

Applications of `QuantumEspressoTools`

In this chapter, I provide several examples of instructions used to build composite `QE` calculations with `QuantumEspressoTools`. The instruction can be executed by the function `execute_serial()` described in Alg.2.6.

3.1 Relax

```
INSTR = INSTRUCTION(
    Dict(`pw.x` => relax_common_settings),
    [ (  common_title * "_iteration_$i",
         `pw.x`,
        ( Dict(  :calc             => "relax",  # calculation type
                 :beta             => beta,     # mixing beta
                 :diag_choices     => diag,     # diagonalization methods
                 "conv_thr"        => 0.01*thr, # scf convergence
                 "etot_conv_thr"   => thr,      # etot convergence
                 "ecutwfc"         => ecutwfc,
                 "ecutrho"         => ecutrho,
                 :updater          => atom_kicker(x,kicker_amplitude),
              ) merge ((i==1) ? (initial_struct) : Dict()) )
       )
       for (i, (thr,kicker_amplitude)) in enumerate(control_list)
    ]
)
```

3.2 Electronic bands

The example codes can be combined and adapted to perform various tasks.

3.2.1 Self-consistent + band structure

```
INSTR = INSTRUCTION(
    Dict( `pw.x`      => scf_common_settings,
          `projwfc.x` => projwfc_common_settings ),
    [(  common_title * "_scf",
        `pw.x`,
        Dict(  "outdir"      => SCF_OUTDIR,
               :calc         => "scf",
               :pw_mode      => "energy",
               :kpoint_mode => "automatic",
               :kpoints      => kpoints,
               "conv_thr"    => conv_thr,
               :updater      => x->copy_xml(x,SCF_OUTDIR,common_title),)
    ),
    (   common_title * "_band",
        `pw.x`,
        Dict(   "outdir"       => SCF_OUTDIR,
               :calc           => "bands",
               :pw_mode        => "bands",
               :beta           => 0.8.*beta,
               :diag_choices   => diag,
               "conv_thr"      => downscale*conv_thr,
               "startingpot"   => "file",
               :updater        => x-> (copy_xml(x,SCF_OUTDIR,common_title)
                                    + find_energy_range(x)),)
    ),
    (   common_title * "_band_projwfc",
        `projwfc.x`,
        Dict(   "prefix"       => common_title,
               "outdir"        => SCF_OUTDIR,
               "filproj"       => common_title * ".proj",)
    )]
)
```

3.2.2 Self-consistent + k-grid energy + wannier90

```
INSTR = INSTRUCTION(
    Dict( `pw.x`             => pw_common_settings,
          `pw2wannier90.x`   => pw2wannier90_common_settings,
          `wannier90.x -pp`  => wannier90_common_settings,
          `wannier90.x`      => wannier90_common_settings,
          `projwfc.x`        => projwfc_common_settings ),
    [(  common_title*"_scf",
        `pw.x`,
        Dict( "outdir"     => SCF_OUTDIR,
              :calc        => "scf",
              :pw_mode     => "energy",
              :kpoint_mode => "automatic",
              :kpoints     => kpoint,
              "conv_thr"   => conv_thr,
              :updater     => x->copy_xml(x,SCF_OUTDIR,common_title),)
    ),
    (   common_title*"_band",
        `pw.x`,
        Dict(  "outdir"       => SCF_OUTDIR,
               :calc          => "bands",
               :pw_mode       => "bands",
               :beta          => 0.8.*beta,
               :diag_choices  => diag,
               "conv_thr"     => downscale*conv_thr,
               "startingpot"  => "file",
                :updater      => x-> (copy_xml(x,SCF_OUTDIR,common_title)
                                  + find_energy_range(x)),)
    ),
    (   common_title*"_band_projwfc",
        `projwfc.x`,
        Dict(  "prefix"       => common_title,
               "outdir"       => SCF_OUTDIR,
               "filproj"      => common_title * ".proj",)
    ),
    (   common_title*"_nscf",
        `pw.x`,
        Dict(  "outdir"       => SCF_OUTDIR,
               :calc          => "nscf",
               :beta          => 0.8.*beta,
               "conv_thr"     => downscale*conv_thr,
               "startingpot"  => "file",
```

```
                :kpoint_mode    => "crystal",
                :kpoints        => kmesh_pl(kp_nscf[1:3]...,0),
                :updater        => x-> (copy_xml(x,SCF_OUTDIR,common_title)
                                    + find_energy_range(x)),)
    ),
    (   common_title*"_projwfc",
        `projwfc.x`,
        Dict(  "prefix"        => common_title,
               "outdir"        => SCF_OUTDIR,
               "filproj"       => common_title * ".proj",
               :updater        => find_window, )
    ),
    (   common_title*"_wannier90",
        `wannier90.x -pp`,
        Dict()
    ),
    (   common_title*"_wannier90",
        `pw2wannier90.x`,
        Dict("outdir" => SCF_OUTDIR,),
    ),
    (   common_title*"_wannier90",
        `wannier90.x`,
        Dict(),
    )]
)
```

3.3 Phonon frequencies

```
INSTR = INSTRUCTION(
    Dict( `pw.x` => pw_common_settings,
          `ph.x` => ph_common_settings, ),
    [(  common_title*"_scf",
        `pw.x`,
        Dict( :calc      => "scf",
              "conv_thr" => conv_thr,
              "ecutwfc"  => ecutwfc,
              "ecutrho"  => ecutrho,
              "disk_io"  => "medium",
              :updater   => get_conv_settings,)
    ),
    (   common_title*"_ph",
        `ph.x`,
        Dict(   "outdir"   => "../$(common_title)_scf",
                "tr2_ph"   => tr2_ph, )
    )]
)
```