PointGroupNRG Manual

Aitor Calvo-Fernández, acalvo049@ehu.eus

August 3, 2024

Contents

1 Overview

This manual describes how to use the `PointGroupNRG` code. Some context and theory are introduced at certain points, but it does not serve as a comprehensive guide to the NRG technique. For a comprehensive review of the NRG method, see Ref. [1]. For a description of the theory and implementation specific to `PointGroupNRG`, see Refs. [2] and [new arxiv], which also contain useful references specific to the applications of symmetry in the NRG.

This manual starts in Section 4 by describing the typical steps in the preparation and execution of an NRG calculation. Section 6 describes the format for the Clebsch-Gordan coefficients that have to be provided in order to use finite point or double group symmetries. Sections 7 and 8 deal with the main functions provided by the code and are organized by describing first the required arguments and then the optional ones, introducing the necessary information along the way. Section 7 covers the `compute_multiplets` function of the `PointGroupNRG.MultipletCalculator` submodule, which constitutes the first step in the setup of a model. Section 8 gives information about the `nrgfull` function of the `PointGroupNRG.NRGCalculator` submodule, which constructs the model and solves it using the NRG.

The functions `compute_multiplets` and `nrgfull` as they are defined in their respective submodules have more optional keyword arguments than those described here. The arguments not described here are in general related to

features in development and changing their value might cause the program to crash.

2 Julia basics

To make the manual self-contained, here we describe the variable types required to use the code and how to construct them, as well as a clarification of the input type notation for functions. Usage examples can be found in the `examples/` directory. Nonetheless, we encourage using the official Julia documentation [7].

The main functions in `PointGroupNRG` have typed arguments, meaning that the variables given as input need to be of the required type for the function to recognize them. Since it is not possible in Julia to define a variable to be of a definite type in global scope as one would do inside a function, *e.g.* `n::Int64=1`, here we describe how to initialize variables to match the desired type.

2.1 Simple variables

- `Int64`: Integer numbers with 64-bit precision. For example,

```
julia> n = 1; # semicolon disables printing
julia> typeof(n) # get type of n
Int64
```

- `Float64`: Real floating point number with 64-bit precision. For example,

```
julia> f = 1.0;
julia> typeof(f)
Float64
```

- `ComplexF64`: Complex floating point number with 64-bit precision. For example,

```
julia> c1 = 1.0 + 0.0im;
julia> typeof(c1)
ComplexF64
julia> c2 = ComplexF64(1.0);
julia> c1==c2 # check equality
true
```

- `String`: For example,

```
julia> s = "PointGroupNRG";
julia> typeof(s)
String
```

- `Function`: A function is a variable type in Julia just as any of the ones mentioned above. For example, $f = x->0.5$, $f(x) = 0.5$ and

```
julia> function f(x)
           return 0.5
       end
julia> f(2.5)
0.5
julia> f(true)
0.5
julia> g = x->0.5 # anonymous function syntax
julia> g(0)==f(0) # equal return values
true
```

Functions can have fixed argument types, each definition of the function with a different set of arguments (with their types) defining a distinct method:

```
julia> f(x::Float64) = "hello"
f (generic function with 1 method)
julia> f(0.5)
"hello"
julia> f(n::Int64) = 1
f (generic function with 2 methods)
julia> f(2)
1
julia> f(true)
ERROR: MethodError: no method matching f(::Bool)
...
```

The argument types can also be parametric:

```
julia> g(x::SF) where {SF<:Union{String,Float64}} = SF
g (generic function with 1 method)
julia> g(1.0)
Float64
julia> g("hello")
String
```

This syntax is used by the functions `compute_multiplets` (7) and `nrgfull` (8.2.2).

2.2 Containers

- `Tuple{T1,T2,...,Tn}`: Fixed-length immutable container with the type determined by the types of its contents. For example,

```
julia> t = (1,"a",1.0);
julias> typeof(t) # get type of t
Tuple{Int64,String,Float64}
```

- `Array{T,N}`: `N`-dimensional array containing elements of type `T`. For example,

```
julia> arr = zeros(ComplexF64,2,3,4);
julia> typeof(arr)
Array{ComplexF64,3}
julia> size(arr) # array dimensions
(2, 3, 4)
```

  Its elements are accessed using the syntax

```
julia> arr[2,3,1]
0.0 + 0.0im
```

- `Vector{T}`: 1-dimensional array containing elements of type `T`. For example,

```
julia> v1 = [1.0, 2.0, 3.0];  # Float64 by default
julia> v2 = Float64[1, 2, 3]; # Float64 conversion
julia> v1==v2 # equal vectors
true
```

- `Matrix{T}`: 2-dimensional array containing elements of type `T`. For example,

```
julia> m1 = [1.0+0.0im 0.0im; 0.0im 1.0+0.0im]
2×2 Matrix{ComplexF64}:
 1  0
 0  1
julia> m2 = ComplexF64[1 0; 0 1];
julia> m1==m2
true
julia> m3 = [1.0;;] # 1x1 matrix syntax
1×1 Matrix{Float64}:
 1.0
```

- `Dict{Tk,Tv}`: Dictionary containing keys of type `Tk` pointing to values of type `Tv`. For example,

```
julia> d = Dict( "a" => [1.0] )
Dict{String, Vector{Float64}} with 1 entry:
  "a" => [1.0]
```

3 Symmetry and irreducible representations

The code offers three symmetry types, one for each symmetry-breaking situation and each with its own irreducible representations:

- Crystal field:

$$G_{\mathrm{UPS}} = U(1)_{\mathrm{C}} \otimes P_{\mathrm{O}} \otimes SU(2)_{\mathrm{S}} \tag{1}$$

  is the combination of the unitary charge symmetry $U(1)_{\mathrm{C}}$ leading to particle conservation, the orbital point group symmetry P_{O}, and the rotational symmetry in spin space $SU(2)_{\mathrm{S}}$ leading to total spin conservation. The irreps are

$$\Gamma_{\mathrm{UPS}} = (N, I, S), \tag{2}$$

  where N is the particle number, I is the point group irrep, and S is the total spin, and they are represented in the code with variables of the type `Tuple{Int64,String,Float64}`. For instance, `(2,"A1g",2.0)` represents the irrep of states with $N = 2$ particles, orbital symmetry $I = A_{1g}$ and total spin $S = 2$; this is the same representation that is going to be used for all symmetry groups: the `Int64` slot is used for the particle number, the `String` slot for the finite (point or double) group irrep, and the `Float64` for the total spin or total angular momentum.

- Spin-orbit coupling:

$$G_{\mathrm{UJ}} = U(1)_{\mathrm{C}} \otimes SU(2)_{\mathrm{OS}} \tag{3}$$

  is the combination of the unitary charge symmetry $U(1)_{\mathrm{C}}$ leading to particle conservation and the spin-orbital rotation group $SU(2)_{\mathrm{OS}}$ leading to the conservation of total angular momentum. The irreps are of the type `(2,"anything",3.5)`, which is the irrep of states with $N = 2$ particles and total angular momentum $J = 3.5$; `"anything"` in the `String` slot just means that the slot is unused, so it can be filled with any `String` and the code will internally handle it as an identity irrep.

- Crystal field and spin-orbit coupling:

$$G_{\mathrm{UD}} = U(1)_{\mathrm{C}} \otimes D_{\mathrm{OS}} \tag{4}$$

  is the combination of the unitary charge symmetry $U(1)_{\mathrm{C}}$ leading to particle conservation and the spin-orbital double group D_{OS}. The irreps are of the type `(2,"A1g",0.0)`, which is the irrep of states with $N = 2$ particles and belonging to the double group irrep $I = A_{1g}$; `0.0` is the identity irrep of angular momentum (spin S or total angular momentum J), which serves to turn off the continuous rotation symmetry $SU(2)$.

When requiring irreps with particle numbers $N = 1$ or $N = 2$, the code follows the following convention:

- $N = 1$: One-particle irreps are completely specified by one quantum number, which the code represents with the indicated type, depending of the symmetry of the model:

  G_{UPS}: Point group irrep I. (The spin of the individual particles is always $S = \frac{1}{2}$.) Represented as a `String`.

  G_{UJ}: Total angular momentum J. Represented as `Float64`.

  G_{UD}: Double group irrep I. Represented as `String`.

- $N = 2$ Two-particle irreps are specified by two quantum numbers, although depending on the symmetry only one is relevant. The variable type is always `Tuple{String,Float64}`, and it stores the information for the various symmetry types as follows:

  G_{UPS}: Point group irrep I and total spin S.

  G_{UJ}: Total angular momentum J. The provided I quantum number will be substituted by the default finite group identity irrep `"A"`.

  G_{UD}: Double group irrep I. The provided S/J quantum number will be substituted by 0.0, the identity irrep of the rotation group.

4 Workflow

The number of steps required for an NRG calculation varies depending on the amount of setup information that is already available. Here we describe all the necessary steps for a full calculation starting from scratch for a model requiring Clebsch-Gordan coefficients for a point or double group. The workflow consists of three main steps:

1. Prepare the directory containing the Clebsch-Gordan coefficients following the format specificed in Section 6. These coefficients will then be read by the functions executed in the next steps. This step is not required for models with total angular momentum conservation, since the Clebsch-Gordan coefficients for the $SU(2)$ spin-orbital symmetry group are computed by the program.

2. Compute the multiplet states for the electronic degrees of freedom of the conduction band and, if necessary, for the impurity (see Section 8.2.2). This is achieved with the `compute_multiplets` function described in Section 7, which calculates the multiplet states and stores them into an appropriate format. This step has to be performed only once for each irrep of the electronic degrees of freedom of a given model, so it is recommended to include it in an independent script separate from step 3.

3. Construct the model and perform the NRG calculation using the `nrgfull` function described in Section 8.2.2. Calculations can be performed also for three intermediate stages giving partial results: two-particle multiplets, the impurity spectrum, and the impurity-shell spectrum (see 8.2.8).

5 Examples

Examples covering all the steps mentioned in 4 are provided in the directory `examples/`. The examples are:

- `krishnamurthy1980`: One-orbital, one-channel Anderson model. See Refs. [8] and [9].
- `infinitecoulomb`: Ionic model of electrons with total angular momentum J (to be fixed by the user) and infinite Coulomb repulsion, leaving only a one-particle multiplet and the empty multiplet. See [6], Ch. 1.9.
- `cubic-e`: Electrons belonging to a subspace belonging to irrep E, which can be viewed as resulting from the splitting of a d level under a crystal field with cubic symmetry O. The standard and ionic Anderson models are used in the script. See [2] for an equivalent E_g model.
- `uranium378`: 3-7-8 model with a non-Fermi liquid ground state, proposed for an uranium impurity. See Ref. [5].

6 Clebsch-Gordan coefficients

Calculations for models with point or double group symmetries require the Clebsch-Gordan coefficients for the group given in the following format: The coefficients are organized according to the reduction of irrep products. If `A` and `B` are the user-given names of two irreps of the finite group G, then their decomposition is

$$\texttt{A} \boxtimes \texttt{B} = \oplus_{\texttt{C}} \texttt{L(C)C}, \tag{5}$$

where `C` are irreps of G and `L(C)` is the number of times `C` appears in the decomposition. On top of the names, the irreps of G must be arbitrarily numbered by the user, starting from 0. If `A`, `B` and `C` are the `a`-th, `b`-th and `c`-th irreps of G, respectively, then the Clebsch-Gordan coefficients of the irrep decomposition of `A`$\boxtimes$`B` must be contained in a file called `axb_AxB.txt`. That file should be organized in paragraphs separated by blank lines, each paragrah containing the Clebsch-Gordan coefficients associated to the subspaces belonging to each irrep in the following format:

```
c C l
( 1 1 | 1 ) = <( A, 1 ; B, 1 | C, 1 )>
...
```

where `l`$= 1, \ldots,$`L` labels the distinct subspaces belonging to `C` that arise in the decomposition of `A`$\boxtimes$`B`, and `<( A, 1 ; B, 1 | C, 1 )>` is meant to be substituted by the value of the Clebsch-Gordan coefficient given in any format that Julia can parse into a complex number. The dots `...` have to be filled with the rest of the Clebsch-Gordan coefficients following the same format,

```
( gA gB | gC ) = <( A, gA ; B, gB | C, gC )
```

where gA, gB and gC are the partner numbers of the irreps A, B and C, respectively. Only the non-zero coefficients are necessary. To compare this with the notation in Refs. [2] and [new arxiv], substitute the irreps (partners) following the rule $\Gamma_A \leftrightarrow$A ($\gamma_A \leftrightarrow$gA), $\Gamma_B \leftrightarrow$B ($\gamma_B \leftrightarrow$gB) and $\Gamma_C \leftrightarrow$C ($\gamma_C \leftrightarrow$gC). The files for all the irrep combinations A$\boxtimes$B with a$\geq$b (to avoid redundancy) must be stored into a directory. An example of Clebsch-Gordan coefficients can be found in examples/clebschgordan/O, which contains the Clebsch-Gordan coefficients for the cubic group O in the appropriate format.

7 Multiplet calculation: compute_multiplets

The function compute_multiplets computes the many-body multiplet states arising from the combination of electrons in states belonging to the same one-electron irrep. If the impurity and conduction electrons occupy states belonging to irreps Γ_1 and Γ_2, for instance, then compute_multiplets has to be run once for Γ_1 and once for Γ_2. This is also the case if there are several multiplet subspaces belonging to Γ_1, for example, because the resulting states are distinguished only by an outer multiplicity label that is taken care of by nrgfull (see Section 8).

The function stores the multiplet states for the irrep G in

```
$(multiplets_path)/$(G)/N$(N).txt
```

N is the number of particles and G and multiplets_path are described in 7.2.1 and 7.2.3, respectively. These files are intended to be read by nrgfull (8). The compute_multiplets function also generates files

```
$(multiplets_path)/$(G)/N$(N)basis.txt
```

containig the basis states expressed as a sequence of N partner numbers for the finite (point or double) group and another sequence for the $SU(2)$ group; if only one group type is used, the numbers of the unused group are substituted by -.

The function is defined as

```
function compute_multiplets(
    symmetry::String ;
    irrep::SF="" ,
    multiplets_path::String="multiplets" ,
    clebschgordan_path::String=""
) where {SF<:Union{String,Float64}}
```

7.1 Necessary arguments

7.1.1 symmetry::String

Determines the type of symmetry. The accepted values are:

- `"pointspin"` or `"PS"`: Uses the symmetry type

$$G_{\mathrm{UPS}} = U(1)_{\mathrm{C}} \otimes P_{\mathrm{O}} \otimes SU(2)_{\mathrm{S}}, \tag{6}$$

  which is the outer direct product of unitary charge symmetry $U(1)_{\mathrm{C}}$ (particle conservation), the orbital point group P_{O}, and spin rotation symmetry $SU(2)_{\mathrm{S}}$ (conservation of total spin).

- `"doublegroup"` or `"D"`: Uses the symmetry type

$$G_{\mathrm{UOS}} = U(1)_{\mathrm{C}} \otimes D_{\mathrm{OS}}, \tag{7}$$

  which is the outer direct product of unitary charge symmetry $U(1)_{\mathrm{C}}$ (particle conservation) and the spin-orbital double group D_{OS}.

- `"totalangularmomentum"`, `"J"`, `"spin"` or `"S"`: Uses the symmetry type

$$G_{\mathrm{UJ}} = U(1)_{\mathrm{C}} \otimes SU(2)_{\mathrm{OS}}, \tag{8}$$

  which is the outer direct product of unitary charge symmetry $U(1)_{\mathrm{C}}$ (particle conservation) and spin-orbital rotational invariance (conservation of total angular momentum).

7.2 Optional keyword arguments

7.2.1 `irrep::SF=""`

One-electron irrep for which to compute the multiplet states. The specification of the irrep changes depending on the value of `symmetry` (7.1.1) according to 3:

- If the value of `symmetry` corresponds to G_{UOS}, then `irrep` must be a `String` and is the name of the name of the point group irrep as it appears in the Clebsch-Gordan coefficient files (see 6).

- If the value of `symmetry` corresponds to G_{UD}, then `irrep` must be a `String` and is the name of the name of the double group irrep as it appears in the Clebsch-Gordan coefficient files (see 6).

- If the value of `symmetry` corresponds to G_{UJ}, then `irrep` must be a `Float` and is the value of the total spin S or total angular momentum J of the electrons, where $\mathbf{J} = \mathbf{L} + \mathbf{S}$.

7.2.2 `clebschgordan_path::String=""`

Absolute or relative path of the directory where the Clebsch-Gordan coefficients are stored (see 6). It must be provided only if `symmetry` (7.1.1) corresponds to G_{UOS} or G_{UD}, as the Clebsch-Gordan coefficients of the rotation symmetry $SU(2)$ are computed by the program.

7.2.3 `multiplets_path::String="multiplets"`

Absolute or relative path of the parent directory where the directories containing the multiplet states for the chosen irreps (see 7.2.1) will be stored. It must be the same for irreps intented to be used in the same NRG calculation. If it does not already exist, it is automatically created.

8 NRG calculation: `nrgfull`

The `nrgfull` function constructs the model from the input and the multiplet states computed by `compute_multiplets` (see 7), and it solves it using the NRG method. It can be used to calculate thermodynamic functions (see 8.2.2), spectral functions (see 8.2.16), and/or impurity projections (see 8.2.20). The function definition is the following (dots `...` in the right-hans side of keyword arguments represent the same types as in the left-hand side):

```
function nrgfull(
    symmetry::String ,
    label::String ,
    L::Float64 ,
    iterations::Int64 ,
    cutoff::IF ,
    shell_config::Dict{SF,Int64} ,
    tunneling::Dict{SF,Matrix{ComplexF64}} ;
    multiplets_path::String="multiplets" ,
    calculation::String="IMP" ,
    impurity_config::Dict{SF,Int64}=Dict{...}() ,
    onsite::Dict{SF,Vector{ComplexF64}}=Dict{...}() ,
    interaction::{Tuple{String,Float64},Matrix{ComplexF64}}=Dict{...}() ,
    spectrum::Dict{ClearIrrep,Vector{Float64}}=Dict{...}() ,
    lehmann_iaj::Dict{ClearTripleG,Array{ComplexF64,4}}=Dict{...}() ,
    until::String="",
    clebschgordan_path::String="" ,
    identityrep::String="" ,
    z::Float64=0.0 ,
    max_SJ2::Int64=10 ,
    channels_dos::Dict{SF,Vector{Function}}=Dict{...}() ,
    minimum_eigenenergy::Float64=0.0 ,
    betabar::Float64=1.0 ,
    spectral::Bool=false ,
    broadening_distribution::String="loggaussian" ,
    spectral_broadening::Float64=0.5 ,
    K_factor::Float64=2.0 ,
    compute_impurity_projections::Bool=false ,
    half_band_width::Float64=1.0 ,
    print_spectrum_levels::Int64=0 ,
```

```
) where {SF<:Union{String,Float64},IF<:Union{Int64,Float64}}
```

8.1 Necessary arguments

8.1.1 `symmetry::String`

See 7.1.1.

8.1.2 `label::String`

Label/name given to the system. It will appear in the names of the output files: thermodynamic functions, spectral functions and impurity projections. Output files are ovewritten by subsequent calculations with the same value of `label`.

8.1.3 `L::Float64`

Discretization parameter Λ. See Ref. [1]. Lower values result in a model closer to the continuum limit $\Lambda \to 1$ but require larger cutoffs, larger values result in convergence for lower cutoffs (see 8.1.5). Large values of Λ usually produce low resolution thermodynamic and spectral functions and can result in artifacts such as spurious oscillations. Oscillations can be removed in many cases by averaging over even and odd step results, which is done automatically, resulting in improved spectral functions (8.2.16) and often completely satisfactory thermodynamic functions (8.2.2. It is possible to go further by averaging over various discretizations (see 8.2.11). In general, it is recommended to use low values $\Lambda \in [2, 3]$ for the spectral functions, while for thermodynamic functions values as large as $\Lambda = 10$ often give satisfactory results.

8.1.4 `iterations::Int64`

Number of iterations in the NRG calculation. Lower temperatures (for thermodynamic function calculations) and lower energies (for spectral function calculations) are reached with more iterations.

8.1.5 `cutoff:IF`

Cutoff imposed on the multiplets: after each iterations, multiplets above the cutoff are discarded. The type of cutoff varies depending on the type of the input `cutoff`:

- If `cutoff` is an `Int64`, *e.g.* `300`, then it specifies the number of multiplets kept at each iteration.
- If `cutoff` is a `Float64`, *e.g.* `7.0`, then it specifies the cutoff energy: multiplets with energies larger than that are discarded.

In both cases, the program tries to avoid breaking accidental degeneracies by checking for a small energy window above the imposed cutoff and keeping also the states within that window.

8.1.6 `shell_config::Dict{SF,Matrix{ComplexF64}}`

It specifies the configuration of the conduction channels. It has the structure

```
Dict(
    G1 => n1,
    G2 => n2,
    ...
)
```

where `G1`, `G2` , etc. are the irreps in the one-electron irep format described in 7.2.1 and `n1`, `n2`, etc. are the number of channels with the symmetry belonging to the corresponding irrep). The multiplet states for each irrep `Gi` must have been calculated previously with `compute_multiplets` (7).

8.1.7 `tunneling::Dict{SF,Matrix{ComplexF64}}`

It specifies the tunneling amplitudes $V(\Gamma_a)_{r_a r_b}$ as a dictionary with the format

```
Dict(
    G1 => amplitudes1,
    G2 => amplitudes2,
    ...
)
```

where `G1`, `G2`, etc. are the irreps Γ_a in the format described in 7.2.1 and `amplitudes1`, `amplitudes2`, etc. are the amplitudes given as a `MatrixComplexF64` with indices r_a, r_b.

8.2 Optional keyword arguments

8.2.1 `multiplets_path::String="multiplets"`

Same as 7.2.3.

8.2.2 `calculation::String="IMP"`

The value of this variable sets whether the impurity is included in the calculation or not, and is relevant in particular for the calculation of thermodynamic functions, which is performed automatically in every case. These are the possible values:

- `calculation="IMP"`: the impurity is included in the calculation, which is necessary for spectral function calculations (8.2.16) and impurity projection (8.2.20) calculations. The thermodynamic functions for the model are stored in the directory `thermodata/`, which is created automatically if it does not previously exist, and are distinguished by containing `imp` in their names. For instance, if `label=="X"` (8.1.2) and `z=="0.0"` (8.2.11), the file would be

```
thermodata/thermo_X_imp_z0.0.dat
```

If a `calculation="CLEAN"` run has been performed (see below), the impurity contribution to the thermodynamic functions is automatically computed by subtracting the `"CLEAN"` results from the `"IMP"` results (see Ref. [1] for more details), and the resulting functions are stored in `thermodata` and distinguished by containing `diff` in their names. Following the previous example, the impurity contribution file would be

```
thermodata/thermo_X_diff_z0.0.dat
```

- `calculation="CLEAN"`: the impurity is excluded for the calculation, leaving only the conduction channels in the model. The resulting thermodynamic functions are stored in `thermodata` and distinguished by containing `clean` in their names. For the same example, the file would be

  ```
  thermodata/thermo_X_clean_z0.0.dat
  ```

  This is typically only used for calculating the impurity contribution to the thermodynamic functions.

8.2.3 `impurity_config::Dict{SF,Int64}`

It defines the degrees of freedom of the impurity for the standard Anderson model (not the ionic model) defined in terms of the on-site energies (8.2.4) and the interaction term (8.2.5). The program assumes that one model or the other is intended based on the value of `spectrum` (8.2.6) and it checks whether the rest of the necessary variables for that model are provided. `impurity_config` follows the same format as 8.1.6 and it also requires the previous calculation of the multiplet states using `compute_multiplets` (7).

8.2.4 `onsite::Dict{SF,Vector{ComplexF64}}=Dict{...}}()`

On-site energy term

$$\sum_a \epsilon(\Gamma_a)_{r_a} c_a^\dagger c_a \tag{9}$$

given in the format

```
Dict(
    G1 => energies1,
    G2 => energies2
    ...
)
```

where `G1`, `G2`, etc. are the irreps Γ_a given in the same format as 7.2.1, and `energies1`, `energies2`, etc. are `Vector`s with index r_a containing the eigenenergies corresponding to the multiplets $m_a = (\Gamma_a, r_a)$. For more information, see Refs. [2] and [new arxiv].

8.2.5 `interaction::Dict{Tuple{String,Float64},Matrix{ComplexF64}}=Dict{...}()`

Parameters of the interaction term

$$\sum_{a,b,c,d} U_{abcd} c_a^\dagger c_b^\dagger c_c c_d \tag{10}$$

defined by the set of parameters $U(\Gamma_u)_{r_u r_v}$ from which the rest are constructed following the procedure described in Ref. [2]. The argument `interaction` contains the parameters $U(\Gamma_u)_{r_u r_v}$ in the format

```
Dict(
    G1 => matrix_elements1,
    G2 => matrix_elements2
    ...
)
```

where `G1`, `G2`, etc. are the two-particle irreps Γ_u, and `matrix_elements1`, `matrix_elements2`, etc. are the matrix elements $U(\Gamma_u)_{r_u r_v}$ for the corresponding irreps with r_u, r_v as indices. The irreps `Gu` are given as `Tuple{String,Float64}`s following the irrep format described in 3

8.2.6 `spectrum::Dict{Tuple{Int64,String,Float64},Vector{Float64}}=Dict{...}()`

It is the spectrum of the impurity in the ionic model, which defines the impurity term in the Hamiltonian (see [new arxiv])

$$H_{\text{imp}} = \sum_i \epsilon(\Gamma_i)_{r_i} |i\rangle\langle i|. \tag{11}$$

The spectrum is provided in the format

```
Dict(
    G1 => energies1,
    G2 => energies2,
    ...
)
```

where `G1`, `G2`, etc. are the many-body irrep of type `Tuple{Int64,String,Float64}` as rescribed in 3, and `energies1`, `energies2`, etc. are the energies of the multiplets $m_i = (\Gamma_i, r_i)$ belonging to the corresponding irrep Γ_i provided as `Vector`s with index r_i.

8.2.7 `lehmann_iaj::Dict{G3,Array{ComplexF64,4}}=Dict{...}()`

Reduced Lehmann amplitudes (matrix elements of the creation operator), used in the ionic model.

$$\langle \Gamma_i, r_i || f^\dagger_{\Gamma_a, r_a} || \Gamma_j, r_j \rangle_\alpha \tag{12}$$

Here `G3` means `NTuple{3,Tuple{Int64,String,Float64}}`. The format of `spectrum` is

```
Dict(
    (G_i,G_a,G_j)_1 => amplitudes_1
    (G_i,G_a,G_j)_2 => amplitudes_2
    ...
)
```

where the irrep tuples `(G_i,G_a,G_j)_1`, `(G_i,G_a,G_j)_2`, etc. represent $(\Gamma_i, \Gamma_a, \Gamma_j)$, and `amplitudes_1`, `amplitudes_2`, etc. are `Array{ComplexF64,4}`s with indices α, r_i, r_a, r_j containing the amplitudes for the irreps specified in the key.

8.2.8 `until::String=""`

Setting `until` to a value different from the default `until=""` causes `nrgfull` to stop at an early stage of the calculation:

- `until="2-particle multplets"`: `nrgfull` runs until it computes the impurity part of the Hamiltonian, after which it prints out information about the 2-particle multiplet states and then stops. This is particularly useful in order to know the symmetry-adapted parameters determining the Coulomb interaction that make up the input variable `interaction` (8.2.5, see Ref. [2])
- `until="impurity spectrum"`: `nrgfull` runs until it computes the impurity spectrum, after which it prints it out and then stops. This is useful for checking the spectrum resulting from a given choice of symmetry-adapted Hamiltonian parameters `onsite` (8.2.4) and `interaction` (8.2.5.
- `until="impurity-shell spectrum"`: `nrgfull` runs until it computes the spectrum of the impurity plus the innermost conduction shell, after which it prints it out and then stops. This is useful for checking the spectrum resulting from the impurity spectrum plus the coupling to the conduction band via tunnneling.

8.2.9 `clebschgordan_path::String=""`

Same as 7.2.2.

8.2.10 `identityrep::String=""`

Identity irrep of the point group or the double group as it appears in the Clebsch-Gordan coefficients given in the directory `clebschgordan_path` (see 7.2.2). It is not necessary if for the G_{UJ} symmetry (see 8.1.1).

8.2.11 `z::Float64::String=""`

Twisting parameter z used for the interleaved discretization scheme following the procedure described in Ref. [3]. Results from calculations with different

z values, both thermodynamic and spectral functions, can be averaged to improve smoothness and remove spurious oscillations arising for large values of Λ (`L`, 8.1.3). The spurious oscillations are already removed to a great extent by averaging over even and odd calculations, which is done automatically (8.2.2 and 8.2.16). Averaging over various z further improves it and allows for an effectively more dense grid of energy values, which can be especially useful for better spectral functions.

The first step in a z-averaged calculations is to call `nrgfull` for all the desired values of z. We can do so using the function `generate_Z` to generate the values of z and then iterating over them, each time calling `nrgfull` for a different `z` value. Then we call the `zavg_thermo` or `zavg_spectral` functions to compute the average thermodynamic or spectral functions, respectively. For instance, we could do

```
# normal input
...

Z = generate_Z(4)
for z in Z
    for calculation in ["CLEAN","IMP"]
        nrgfull(
            ...;
            calculation=calculation,
            z=z
            ...
        )
    end
    zavg_thermo(label, Z)
end
```

for computing z-averaged thermodynamic functions, or

```
# normal input
...
Nz = 4
Z = generate_Z(Nz)
for z in Z
    nrgfull(
        ...;
        spectral=true,
        z=z
        ...
    )
    end
    zavg_spectral(label, Z; orbitalresolved_number=2)
end
```

where the `orbitalresolved_number` variable tells `zavg_spectral` how many tunneling electron multiplets are there, and therefore which spectral files have to be taken into account when averaging (8.2.16).

One of the main advantages of z-averaged calculations is that they can are completely independent from one another and can therefore be performed in parallel. This is very simple in Julia, and it can be done as

```
using Distributed
Nz = 4
Z = generate_Z(Nz)
addprocs(Nz)
@everywhere begin
    # define input in all the processors here
    label=$label
    ...
end
@sync @distributed for z in Z
    nrgfull(
        ...;
        z=z,
    )
end
```

An example of a distributed z-averaged calculation is provided in `examples/uranium378` (5).

8.2.12 `max_SJ2::Int64=10`

Maximum value of twice the total spin $2S$ (for G_{PS} or G_{UJ}) or twice the total angular momentum $2J$ (for G_{UJ}) that is expected to appear in the NRG calculations (see 8.1.1). This is used for computing sums over Clebsch-Gordan coefficients, which is done at the beginning of the calculation. If a value of $2S$ or $2J$ larger than `max_SJ2` if found, the program will give an error.

8.2.13 `channels_dos::Dict{SF,VectorFunction}=Dict{SF,VectorFunction}()`

Density of state (DOS) functions $\rho(\Gamma_b;\epsilon)_{r_b}$ of the conduction electron channel multiplet (Γ_b, r_b). In the code, the density of states is defined as (i) carrying the energy-dependence of the hybridization,

$$\Delta(\Gamma_b;\epsilon)_{r_b} := \pi\rho(\Gamma_b;\epsilon)_{r_b} V(\Gamma_b)^2_{r_b r_b} \tag{13}$$

and (ii) normalized as

$$\int_{-D}^{D} d\epsilon \rho(\Gamma_b;\epsilon)_{r_b} = 1, \tag{14}$$

where D is the half band-width.

For instance,

```
channels_dos_1 = Dict(
    "E" => [x->0.5]
)
channels_dos_2 = Dict(
    "E" => [x->1.0]
)
```

define two equal constant DOS functions because the code automatically normalizes the density of states functions according to Eq. 14. The discretization is performed for general DOS functions following the procedure described in Ref. [4].

8.2.14 `minimum_eigenenergy::Float64=0.0`

If the largest eigenergy remaining after applying the multiplet cutoff according to 8.1.5 is lower than `minimum_eigenenergy`, the number of multiplets kept is increased until reaching the eigenenergy `minimum_eigenenergy`.

8.2.15 `betabar::Float64=1.0`

Parameter $\bar{\beta}$ used in thermodynamic calculations. When calculating thermodynamic functions, the Boltzmann factors $\exp(-\beta E_N)$, where $\beta = (k_B T)^{-}1$ and E_N is an eigenenergy in the N-th NRG step, are substituted by $\exp(-\bar{\beta}\bar{E}_N)$, where

$$\bar{\beta} = \omega_N \beta = \frac{\omega_N}{k_B T} \tag{15}$$

is a constant that fixes the ratio between the energy scale ω_N of each iteration and the temperature T, thereby defining the temperature scale

$$T_N = \frac{\omega_N}{k_B \bar{\beta}} \tag{16}$$

associated to each iteration N. For more detailed information, see Ref. [1].

8.2.16 `spectral::Bool=false`

If `spectral==true` and `calculation=="IMP"` (8.2.2), `nrgfull` computes the zero-temperature orbital-resolved spectral functions

$$A_{\Gamma_a, r_a}(\omega) = \sum_g |\langle e| f^\dagger_{\Gamma_a, r_a} |g\rangle|^2 \delta(\epsilon - (\epsilon_e - \epsilon_g)) \tag{17}$$

which it then stores in files

```
spectral/spectral\_<label>\_z<z>\_o<o>.dat
```

where `<label>` is the value of `label` (8.1.2), `<z>` is the value of `z` 8.2.11, and `<o>` is a number assigned by the code to each orbital multiplet Γ_a, r_a. Which orbital multiplet corresponds to each number is indicated in the file header.

The spectral function contained in the file is obtained by averaging the spectral functions for the even and odd iterations [REF]; information about the latter is stored in the files

```
spectral/spectral\_<label>\_z<z>\_o<o>_even.dat
spectral/spectral\_<label>\_z<z>\_o<o>_odd.dat
```

8.2.17 `broadening_distribution::String="loggaussian"`

Broadening kernel used to obtain continuous spectral functions from the set of delta peaks calculated using Eq. 17. Three options are available:

- `broadening_distribution="loggaussian"`:

$$\delta(\epsilon-(\epsilon_e-\epsilon_g)) \to \frac{1}{\eta\sqrt{\pi}|\epsilon_e-\epsilon_g|}\exp\left(\frac{\log^2|\frac{\epsilon}{\epsilon_e-\epsilon_g}|}{\eta^2}-\frac{\eta^2}{4}\right) \tag{18}$$

- `broadening_distribution="gaussian"`:

$$\delta(\epsilon-(\epsilon_e-\epsilon_g)) \to \frac{1}{\sqrt{\pi}\eta}\exp\left[-\left(\frac{\epsilon-(\epsilon_e-\epsilon_g)}{\eta}\right)^2\right] \tag{19}$$

- `broadening_distribution="lorentzian"`:

$$\delta(\epsilon-(\epsilon_e-\epsilon_g)) \to \frac{1}{2\pi}\frac{\eta}{(\epsilon-(\epsilon_e-\epsilon_g))^2+\eta^2} \tag{20}$$

In all cases, η is the broadening fractor fixed by the argument `spectral_broadening` (8.2.18).

8.2.18 `spectral_broadening::Float64=0.5`

Broadening factor η to be used in the broadening kernel (see 8.2.17).

8.2.19 `K_factor::Float64=2.0`

When calculating the spectral functions $A_{\Gamma_a,r_a}(\omega)$ (see Eq. 17), each NRG step is used to obtain the value $A_{\Gamma_a,r_a}(\omega)$ at an energy $K\omega_N$, where ω_N is the energy scale of the N-th step and K is the quantity fixed by `K_factor` (see Ref. [1]).

8.2.20 `compute_impurity_projections::Bool=false`

If `compute_impurity_projections==true`, `nrgfull` computes the thermodynamic weights of the impurity projections,

$$P_{\Gamma_i,r_i}(T_N) = \frac{\mathrm{Tr}\{\hat{P}_{\Gamma_i,r_i}e^{-\beta_N\hat{H}}\}}{\mathcal{Z}}, \tag{21}$$

where $\mathcal{Z}$ is the partition function and $\hat{P}_{\Gamma_i,r_i}$ are operators that project the impurity-conduction states $|u\rangle_N$ obtained in the N-th NRG step onto the impurity multiplet Γ_i, r_i,

$$\hat{P}_{\Gamma_i,r_i} = \sum_{\gamma_i} |\Gamma_i, \gamma_i, r_i\rangle_{\mathrm{imp}} \hat{I}_{\mathrm{con}} \langle \Gamma_i, \gamma_i, r_i|_{\mathrm{imp}}, \tag{22}$$

where $|\Gamma_i, r_i, r_i\rangle_{\mathrm{imp}}$ are impurity eigenstates belonging to the impurity multiplet Γ_i, r_i and $\hat{I}_{\mathrm{con}}$ is the identity operator of the conduction electron subspace. They are computed in the same way as other thermodynamic quantities (see 8.2.2 and Ref. [1]).

The results are stored in

```
impurityprojections/imp_proj_<label>_z<z>.dat
```

The directory `impurityprojections` is automatically created if it does not exist already. The first column of the file contains the NRG iteration, the second column contains the temperature, and subsequent columns contain $P_{\Gamma_i,r_i}(T_N)$ for the impurity multiplets Γ_i, r_i (information in the header of the file).

The information provided by P_{Γ_i,r_i} is useful in itself in order to obtain information about the impurity state at all temperatures, but it can also be used to compute the thermodynamic function of an impurity operator that only depends on the impurity multiplet: if $\hat{X} = X(\Gamma_i, r_i)\hat{P}_{\Gamma_i,r_i}$ is the operator, then

$$X(T_N) = \frac{\mathrm{Tr}\{\hat{X}e^{-\beta_N \hat{H}}\}}{\mathcal{Z}} = \sum_{\Gamma_i,r_i} X(\Gamma_i, r_i) P_{\Gamma_i,r_i}. \tag{23}$$

This is not implemented in `PointGroupNRG`, so it is left to to the user as a postprocessing step.

8.2.21 `half_band_width::Float64=1.0`

Half band-width D used in the NRG calculation (see Ref. [8]).

8.2.22 `print_spectrum_levels::Int64=0`

If `print_spectrum_levels!==0`, the lowest M multiplet eigenenergies are printed after each NRG step, where M is the value of `print_spectrum_levels`.