

2B_MatrixElements suite Documentation

Miguel de la Fuente

February 29, 2024

Contents

1	Introduction	2
2	Object oriented Matrix Elements	4
2.1	Wave functions	4
2.2	Matrix elements	4
3	Calculation over valence spaces, TBME_Runner calculator.	7
3.1	Input File (xml)	7
3.1.1	Interaction_Title	7
3.1.2	Output_Parameters	7
3.1.3	SHO_Parameters	8
3.1.4	Valence_Space	9
3.1.5	Core	9
3.1.6	Force_Parameters	10
4	Implemented Forces. Parameters	12
4.1	Central (JT scheme)	12
4.2	Update for Central interaction, introduction of exchange operators.	13
4.3	Coulomb (J scheme)	13
4.4	PotentialSeries (JT scheme)	13
4.5	Brink_Boeker (JT scheme)	13

4.6	Density_Dependent (JT scheme)	14
4.7	Density_From_File (J scheme)	15
4.8	DensityFiniteRange (JT scheme)	15
4.9	Kinetic_2Body (JT scheme)	17
4.10	SpinOrbitShortRange (JT scheme)	17
4.11	SpinOrbit and Tensor (JT scheme)	18
4.11.1	Tensor S_{12} interaction with Exchange operators: TensorS12 (JT scheme).	18
4.11.2	Spin-Orbit finite range with Exchange operators: SpinOrbitFiniteRange , SpinOrbitSquaredFiniteRange and OrbitalMomentumSquared	19
4.12	SDI / MSDI Surface Delta Interaction (JT scheme)	19
4.13	Multipole_Delta Interaction (JT scheme)	19
4.14	Multipole_Moment Interaction (JT scheme)	20
4.15	SkrymeBulk_JTScheme Interaction (JT scheme)	20
4.16	Force_From_File	21
5	Debbuging and Decomposition of Matrix elements (XLog class)	22
6	Comparing matrix elements	24
6.1	Comparing two files in JT scheme for testing. MatrixElementFilesComparator	24
6.2	Comparing a (reduced valence space) sets of matrix elements by plotting their relative differences.	26

1 Introduction

This Python suite allows the development of two body interactions in the simple harmonic oscillator basis (SHO), it's programmed in the object-oriented paradigm, which allows us the reuse of methods, the specification based on features (Coupling scheme, isospin symmetry, center of mass decomposition, ...) and simplify the understanding and development of new matrix elements.

The suite has an implementation of a matrix element calculator to obtain interactions over valence space, getting the matrix elements in a suitable format for Antoine and Taurus codes¹. The calculator is called **TBME_Runner** and uses xml input, a easy format to understand and useful to automate.

¹[Antoine] shell model calculations: <http://www.iphc.cnrs.fr/nutheo/code-antoine/menu.html> code
[Taurus] HFB with symmetry restoration code: <https://github.com/project-taurus>

It is also equipped with a *tree*-based debugger, which writes nested lines based in the order of apparition. This is especially useful to decompose the matrix elements in the different levels since they are usually based on series and coefficients, letting us verify the implementation of the matrix elements (also didactic).

2 Object oriented Matrix Elements

2.1 Wave functions

One body wave functions are based on the SHO base: $n \geq 0$, $l \geq 0$, $m_l \leq |l|$, can also label as proton/neutron with the third component of isospin $m_t = (p : +1, n - 1)$ (optional, default is 0). `QN_1body_radial`. There is also jj coupling, where there is the $j = l + s$ variable and its third component $m \leq |j|$.

Then, based on single particle wave functions, we can construct two body wave functions that we couple to an angular momentum scheme. Particle restrictions for the 2B wave function is now mandatory. When using the L `QN_2body_L_Coupling`, LS `QN_2body_LS_Coupling`, J `QN_2body_J_Coupling` scheme, particle labels are necessary, so they inherit from `_ParticleWaveFunction` to verify arguments

2.2 Matrix elements

Matrix elements must be directly related to the type of wave functions, the developer should take care of the complexity of a certain type of coupling or the antisymmetrization.

All matrix elements have four abstract methods to be implemented:

1. **Interaction Parameter Setter:** `setInteractionParameters(**kwargs)` It is a *class-method* (and must be decorated as such) that fix the interaction parameters for an interaction, both for checking input arguments and manage them. *keyword arguments* usually include the SHO nuclear parameters (the b length, A mass, ... and stored in attribute `PARAMS_SHO`) and interaction parameters (potential types, indexes, characteristic lengths and constants, ... stored in attribute `PARAMS_FORCE`).²

Notice that matrix elements to be called from `TBME_Runner` must work with the `io_manager.CalculationArgs` input manager (xml reading), when a new force is introduced, set its parameters in `helpers.Enums` and add the new class in `matrix.elements.__init__.switchMatrixElementType` for the automation (details for this part in Input File section 3).

2. **The Constructor:** `__init__(bra, ket, run_it=True)` verifies the Input arguments and properties of the wave function by `__checkInputArguments(bra, ket)` (i.e, the wave functions for J scheme must inherit from `_ParticleWaveFunction`) and evaluate its value for the argument `run_it` by calling `_run()`, the constructor is called always after the `setInteractionParameters` method.
3. `_run()` This method evaluates the value of the matrix element, this value is accessible from the object *property* "value", which calls the run method if the instance has not been calculated.

This method is protected, so must not called from outside. It intended to do the primary actions of the calculation such as the two-body normalization, the global antisymmetrization, and other main subroutines.

²Remember to refresh in this part the arguments, since further calls of the interaction will only overwrite previous declarations and optional arguments might lead to conflict. I.e, do `PARAMS_FORCE={}` before managing the parameters to start from scratch the interaction but keeping the `PARAMS_SHO` arguments.

4. **Arguments checker:** `__checkInputArguments(bra, ket)`, we already mentioned, but it is recommended to be implemented to prevent issues.

This is one example for calling an individual matrix element (Density-dependent matrix element for context)

$$\langle n_a l_a j_a \ n_b l_b j_b | V(x_0, C, \alpha, A) | n_c l_c j_c \ n_d l_d j_d \rangle_{JT} = \langle 0d_{5/2} \ 0d_{3/2} | V(1, 1380, 1/3, 4) | 0d_{3/2} \ 0d_{5/2} \rangle_{2,1}$$

and prompt the value on console.

```
## 1. Define the keyword arguments (using Enum enumerations)
kwargs = {
    # wave function parameters
    SHO_Parameters.A_Mass      : 4,
    SHO_Parameters.b_length    : 1.4989,
    SHO_Parameters.hbar_omega   : 18.4586,

    # central general parameters
    CentralMEParameters.potential : PotentialForms.Power,
    CentralMEParameters.mu_length : 1,
    CentralMEParameters.constant  : 1380,
    CentralMEParameters.n_power   : 0,

    # density dependent specific parameters
    DensityDependentParameters.alpha : 1/3,
    DensityDependentParameters.x0 : 1,
    DensityDependentParameters.constant : 1
}

## 2. define the two body wave functions for bra and ket
J = 2
T = 1
bra_ = QN_2body_jj_JT_Coupling(QN_1body_jj(0,2,3),
                                QN_1body_jj(0,2,5), J, T)
ket_ = QN_2body_jj_JT_Coupling(QN_1body_jj(0,2,3),
                                QN_1body_jj(0,2,5), J, T)

# This saves the calculation data as tree for debbuging (see section 3)
DensityDependentForce_JTScheme.turnDebugMode(True)

## 3. set the previous interaction parameters.
DensityDependentForce_JTScheme.setInteractionParameters(**kwargs)

## 4. Evaluate the matrix element by instance it.
me = DensityDependentForce_JTScheme(bra_, ket_)

## 5. access to the numerical value
print("me: ", me.value)

# this saves the log as a xml tree
me.saveXLog()
```

Almost all of my implemented matrix elements implicitly use the LS coupling to evaluate the radial part in the *L-Scheme*, but Antoine/Taurus use exclusively the *JT-Scheme* or the *J-Scheme*. Then all matrix elements evaluated for that input have to couple from *J-Scheme* to *LS-Scheme* via *9j symbols* applied on the wave functions. That is why all elements used the `TBME_Runner` inherit from `_TwoBodyMatrixElement_JTCoupled`, they share the decoupling method `_LS_recoupling_ME()` (previously the `_non_antisymmetrized_ME()`) and differ in an internal polymorphic method `_LScoupled_MatrixElement()` called by the former.

Examples (the unmentioned parent classes just implement a more general form of the matrix element, they implement the common coefficients/methods):

1. `CentralForce_JTScheme(CentralForce, _TwoBodyMatrixElement_JTCoupled)`,
2. `CoulombForce(CentralForce, _TwoBodyMatrixElement_JTCoupled)`,
3. `SpinOrbitForce_JTScheme(_TwoBodyMatrixElement_JTCoupled, SpinOrbitForce)`

Additionally, the antisymmetrization can be done explicitly by exchanging the ket quantum numbers and then evaluating again the matrix element non antisymmetrized function (`_LScoupled_MatrixElement()`).

$$\begin{aligned}\langle ab|V|cd\rangle_{JT}^{as} &= \langle ab|V|cd\rangle_{JT} - (-)^{j_c+j_d+J+T+1} \langle ab|V|dc\rangle_{JT} \\ \langle ab|V|cd\rangle_J^{as} &= \langle ab|V|cd\rangle_J - (-)^{j_c+j_d+J} \langle ab|V|dc\rangle_J\end{aligned}$$

This process is implemented in the `_run()` for the classes `_TwoBodyMatrixElement_Antisym_JTCoupled` (`_TwoBodyMatrixElement_JTCoupled`) (and JT).

Nevertheless, besides `_TwoBodyMatrixElement_JTCoupled` in principle only evaluate the two-body normalization and call directly the non-antisymmetrized function, many interactions have symmetries in the wave functions and their internal coefficients to bring the exchanged part without evaluating two times the `_non_antisymmetrized_ME()` method. A smarter deduction can be implemented ³ in the internal LS-method or overwriting the `_run()` and splitting the running time in half.

These different ways of computing the antisymmetrization are mediated by another overwritable method inside the LS recoupling: `_antisymmetrized_LS_element()`. *9j symbols* change by a phase when two rows are permuted. For the ket symbol it implies just the addition of another phase, so it is not necessary to evaluate explicitly the whole element (reduce recoupling time [5/10/21]).

$$\begin{Bmatrix} l_c & s_c & j_c \\ l_d & s_d & j_d \\ \Lambda & S & J \end{Bmatrix} = (-)^{j_c+j_d+J+s_c+s_d+S+l_c+l_d+\Lambda} \begin{Bmatrix} l_d & s_d & j_d \\ l_c & s_c & j_c \\ \Lambda & S & J \end{Bmatrix}$$

This method calls directly the function for the LS matrix element (internally antisymmetrized) or evaluates that method for the direct and the exchanged one in LS scheme.

³It usually is a factor 2 or canceling as a phase depending on the quantum numbers

3 Calculation over valence spaces, TBME_Runner calculator.

The program is designed to produce matrix elements with a suitable format for the programs Taurus and Antoine. That is *JT-Scheme* and *J-Scheme* with particle-labeled matrix elements explained in section 2.2. There are five parts in the XML input⁴ managing the output settings, the SHO wave function, the valence space definition, and the interaction parameters.

3.1 Input File (xml)

XML trees store the data as stings in a leaf tag as attributes or text: `<tag attr1='1' attr='a' ...>text</tag>`. Every tag name inserted has a mandatory string and so the attribute keys (also the values in case the parameter given is string type or for certain value schema such as integer range, boolean, ...), are necessary for the automation and we need to fix them in `helpers.Enums`, some of them might be optional.

3.1.1 Interaction_Title

Just the name of the output file, has two attributes:

1. **name** (mandatory) `<string>`. Set the file title (not the filename) for the two body matrix elements.
2. **details** (optional) `<string>`. A single-line description is to be after the m.e. title in the first line of the Antoine output file.

```
<Interaction_Title name="TBME_Runner" details=""/>
```

3.1.2 Output_Parameters

This sets the output filename and its Antoine/Taurus format. This entry has only leaves with text, all without attributes.

1. **Output_Filename** (mandatory) `<string>`. Set the file name for the two body matrix elements. Don't put the extension (it will be given by the next argument)
2. **Hamil_Type** (mandatory) `<int = 1:4>`. Taurus accepts different types of matrix elements, the argument sets them accordingly:
 - 1: *JT-scheme* where single-particle energies of protons and neutrons are equal. "sho" extension output, also valid for Antoine.
 - 2: Not implemented
 - 3: Not implemented

⁴There is also a *json* input parser from an early version of the program, but this implementation has been for long untested, and should be considered as a deprecated input format.

4: *J-scheme* where states are particle-labeled for protons and neutrons for the six two body elements⁵. Produces a "sho" extension output with the SHO arguments and states, that the actual interaction matrix elements are in another "2b" extension with the same filename. This format uses another form of integer SHO label for the states, where l can be greater than 10.

Hamil_type 1 can only be used when the interactions are isospin symmetric, this specification is set in the actual matrix element class in the `COUPLING` class attribute, define them using `Enums.CouplingSchemeEnum` enumeration. Type 4 accepts all cases (i.e Coulomb interaction.)

3. **COM_correction** (optional) <int = 0:1>. This is an optional argument to include the effect of the center of mass correction for the elements. COM_correction = 0 by default.

```
<Output_Parameters>
  <Output_Filename>dd_SPSP</Output_Filename>
  <Hamil_Type>1</Hamil_Type>
  <COM_correction>0</COM_correction>
</Output_Parameters>
```

3.1.3 SHO_Parameters

Define the spherical oscillator parameters, and are the A nucleus mass, the $\hbar\omega$ value, and/or the b length.

The requirements just need to define an oscillator length, for example, if A is given and the others don't, the program uses an empirical relationship to fix them. Given only $\hbar\omega$ we fix b (if the interaction depends on A it must raise an error) and vice-versa. $\hbar\omega$ is used to get the single-particle energies according to the spherical oscillator formula in case these are not given.

$$\hbar\omega = 41A^{-1/3} \quad b = \frac{\hbar c}{\sqrt{m_{Nucleon}\hbar\omega}} \quad (1)$$

Where the constant definitions can be seen in `helpers.Helper.Constants` enumeration. The parameters are independent when manually fixed.

1. **A** <int >, default 0
2. **hbar_omega** <float positive >, units MeV, default depending on A, and if b exists, it is left as 0.
3. **b.length** <float positive >, default depending on the previous parameters: If $\hbar\omega$ is not given, b comes from the formula $\hbar\omega(A)$, if $\hbar\omega$ is given, b length comes from the $b(\hbar\omega)$ formula.

Example (comments on xml follow <!-- ... until -->):

```
<SHO_Parameters>
  <!-- A_Mass (int) of the nucleus mandatory, hbar_omega and b (floats)
       are not required parameters, define values only for explicit
```

⁵From left (0) to right (5): *pppp*, *pnpn*, *pnpn*, *npnp*, *nnnn*


```

        setting.
-->
<A_mass>16</A_mass>
<hbar_omega units='MeV'>18.4586</hbar_omega>
<b_length units='fm'>1.4989</b_length>
</SHO_Parameters>

```

3.1.4 Valence_Space

This part defines the single-particle states for the TBME to be evaluated and optionally their energies. As said before, the single particle notation for Antoine implies $l < 10$, but ab initio methods could need matrix elements with $l > 10$ and the index.

Index *spss*:

$$l < 10 \quad \rightarrow \quad spss = 1000n + 100l + 2j \quad (2)$$

$$l \geq 10 \quad \rightarrow \quad spss = 10000n + 100l + 2j \quad (3)$$

To read in each case we set `l_great_than_10`, internally the code read in $l > 10$. Each state is introduced as a leaf in the form `<Q_Number sp_state='int' sp_energy(optional)='float'/>`, The single particle energy is optional and if it is not given (or given as a null value: `sp_energy=''`), it is set by the SHO energy:

$$E_{nl} = \hbar\omega(2n + l + 3/2)$$

The energies are optional, but when given, all valence space energies must be set (mixing default and explicit values raises an error.)

The two examples below evaluate the same valence space $1p_{3/2}$, $0f_{5/2}$, $1p_{1/2}$ and $0g_{9/2}$ read with $l > 10$ and not.

```

<Valence_Space l_great_than_10='True'>
  <Q_Number sp_state='10103' sp_energy='' />
  <Q_Number sp_state='305' sp_energy='' />
  <Q_Number sp_state='10101' sp_energy='' />
  <Q_Number sp_state='409' sp_energy='' /-->
</Valence_Space>

```

```

<Valence_Space l_great_than_10='False'>
  <Q_Number sp_state='1103' sp_energy='' />
  <Q_Number sp_state='305' sp_energy='' />
  <Q_Number sp_state='1101' sp_energy='' />
  <Q_Number sp_state='409' sp_energy='' /-->
</Valence_Space>

```

3.1.5 Core

The core part can be taken into account for Antoine and Taurus code, it set the 4th line in the output. This part was not developed in the program. This is an example to set it:

```

<Core>
  <protons>0</protons>
  <neutrons>0</neutrons>
  <innert_core name='' />
</Core>

```

All arguments are optional and can be omitted by setting `<Core/>` in the file, when not given, the default is `Z=N=0` and `<innert_core name='NO' />`.

3.1.6 Force_Parameters

This is the main part of the input and sets the required parameters for the designed matrix element classes. The internal parameters for each force can be created freely but first, there are some things in this section:

1. **Forces are additive and can be selected multiple times**, we can set as many forces as many times as we want to get a linear combination of interactions. For example, Gogny D1S force is a combination of a Brink-Boeker, a short-range approximation of the spin-orbit interaction, a Coulomb interaction, and a density-dependent term. Then we define the parameters of each individual force in the section:

```

<Force_Parameters>
  <Brink_Boeker> ... </Brink_Boeker>
  <Coulomb/>
  <SpinOrbitShortRange> ... </SpinOrbitShortRange>
  <Density_Dependent> ... </Density_Dependent>
</Force_Parameters>

```

Another use is to sum simple interactions, for example, several central potentials

$$V(r) = -100e^{-r^2/1.5} + 420/r - 60$$

then (parameter details in the next section):

```

</Force_Parameters>
  <Central>
    <potential name='gaussian' />
    <constant value='-100.0' units='MeV' />
    <mu_length value='1.5' units='fm' />
  </Central>
  <Central>
    <potential name='coulomb' />
    <constant value='420.0' units='MeV' />
    <mu_length value='1.5' units='fm' />
  </Central>
  <Central>
    <potential name='power' />
    <constant value='-60.0' units='MeV' />
    <mu_length value='1.0' units='fm' />
    <n_power value='0' />
  </Central>
</Force_Parameters>

```

Notice that every entry in the Force parameters node will be evaluated separately, as we see in the Matrix Element running section, this means to evaluate all valence space LS recoupling and inner calculation and then go for the next. Smart choice/design of the matrix elements can save us time. For example, if we want to describe some potential as a series of 30 Gaussians, we don't need to create 30 Central individual instances of Gaussians, there is another Interaction called `GaussianSeries` that implements as many different Gaussians for each LS matrix element by the Talmi-Moshinsky method, being the total computation time slightly longer than a simple Central interaction.

2. **Forces can be in set in the file without executing it** by including the boolean attribute `active='False'` in the force parent node. Its default value is 'True' but might save some time when testing or running by not needing to remove the forces.
3. As always, a brief description can be added as a comment.
4. Forces don't necessarily have to have parameters. I.e, Coulomb Interaction is predefined as a $V(r) = \delta(pppp)1.4523/r$, so we just need to put `<Coulomb/>` or `<Coulomb active='True'>` in the section.
5. **Force parameters, attributes to read, and the TBME_Runner selector entry must be defined when creating a matrix element for the automation.** To ensure the maintainability, cleanliness, and flexibility of the program, for every new interaction implemented we need to:

- Include the interaction name's input to be called in the `helpers.Enum.ForceEnum`. These names don't have to be the same as the class that is executed.

```
class ForceEnum(Enum):
    Central = 'Central'
    Coulomb = 'Coulomb'
    {Name for the code to use} = '{Name for the input}'
```

- Connect that name with the class in the matrix element module `matrix_element.__init__.py` in the function `switchMatrixElementType(force)`.
- Each force branch has its parameter entries, the tag must be defined in a class for these arguments. See f.e. `helpers.Enums.CentralMEParameters`, `BrinkBoekerParameters`, etc, (name format is `{ForceName}Parameters(Enum)`).

These enumerations have to be connected with their force for the code to verify while parsing the input file, which is done in the dictionary `helpers.Enums.ForceVariablesDict`

```
ForceVariablesDict = {
    ForceEnum.Brink_Boeker : BrinkBoekerParameters,
    ForceEnum.Central : CentralMEParameters,
    ...
```

- Fix the attribute keys that appear for every previous variable in the force branch. Any element with attributes is passed to the class setup method as a dictionary, creating (or reusing) an enumeration to read them. The enumeration `helpers.Enums.AttributeArgs` and its inner "subenumeration" `AttributeArgs.ForceArgs` serve for that implementation (If the attributes don't affect the parsing, they can be fixed in another class). The point of doing this is to standardize the code and the parameters for the interaction setup. Many variables share the attributes "name", and "value", ... reuse them if possible to avoid absurd multiplication of the enumerations.

- When the values of the arguments are specific string, standardize the valid words in another enumeration (as we did in `ForceEnum`, see f.e. `helpers.Enums.PotentialForms`):

```
class Units(Enum):
    MeV = 'MeV'
    fm = 'fm'
```

6. Use *JT-scheme* preferably than the particle labeled for the TBME implementation, the latter implies the output to be `hamil_type=3, 4` since it's meant to be used with isospin symmetry-broken interactions (such as Coulomb or other *ab-initio* interactions.).

4 Implemented Forces. Parameters

Updated (12/8/2021) Updated (10/2/2023) Updated (29/2/2024)

4.1 Central (JT scheme)

Central force implements some common potentials in the *JT-scheme* that depend on the relative distance. The available potentials are in `helpers.Enums.PotentialForms` and the main arguments are in `helpers.Enums.CentralMEParameters`: the factor `constant` (in MeV), an intrinsic length `mu_length` (in fm), and an index `n.power` for some potentials. Notice that,

Name for Input	Potential $V(r)$	Mandatory parameters
gaussian	$V_0 \exp(-(r/\mu)^2)$	<code>constant, mu_length</code>
exponential	$V_0 \exp(-r/\mu)$	<code>constant, mu_length</code>
coulomb	$V_0 \mu/r$	<code>constant, mu_length</code>
yukawa	$V_0 \exp(-r/\mu)/(r/\mu)$	<code>constant, mu_length</code>
power	$V_0 (r/\mu)^n$	<code>constant, mu_length, n.power</code>
gaussian_power	$V_0 \exp(-(r/\mu)^2)/(r/\mu)^n$	<code>constant, mu_length, n.power</code>

Table 1: Central Potential Forms and their parameters

"coulomb" refers to the inverse potential, for the real proton-proton repulsion interaction uses force Coulomb. For a constant potential use "power" and set $n=0$. These central forces use the method given by Talmi-Moshinsky ⁶, where the Talmi radial integral is defined for the code as:

$$I_p = \frac{\sqrt{2}b^2}{\Gamma(p + \frac{3}{2})} \int_0^\infty dr \left(\frac{r^2}{2b^2} \right)^{p+1} V(r) \exp\left(-\frac{r^2}{2b^2}\right) \quad (4)$$

this definition differs slightly from the reference, it takes into account the oscillator length and the factor $\sqrt{2}$ due to the relative radial coordinate of the wave functions in which the integral is evaluated. Use this definition if you want to use the implemented Moshinsky transformation.

```
<Central active='True'>
  <potential name='gaussian' />
  <constant value='-100.0' units='MeV' />
  <mu_length value='1.4' units='fm' />
</Central>
```

⁶T. Brody, M. Moshinsky Tables of transformation Brackets for nuclear shell model calculations (1960)

4.2 Update for Central interaction, introduction of exchange operators.

It is more useful to generalize the central interaction to evaluate traditional exchange operators as appeared in the next **Brink_Boeker** interaction. These are Wigner (none), Barlett (spin), Heisenberg (isospin), and Majorana (orbital) exchanges.

$$V = V(r) (W + BP^\sigma - HP^\tau - MP^\sigma P^\tau) \quad (5)$$

It can be given the attribute **constant**, in case of no exchange operators given,

4.3 Coulomb (J scheme)

Coulomb is the proton-proton electrostatic repulsion, it's here because the interaction breaks isospin and only acts on proton-proton states, when used, matrix elements must be in *J-scheme*. Their parameters are fixed internally.

```
<Coulomb active='True'>
</Coulomb>
```

4.4 PotentialSeries (JT scheme)

Potential series is an extension of the central force for efficiency. For example, We can mimic some potentials based on a series of Gaussians.

$$V(r) \approx \sum_{i=1}^N V_0^i e^{-r^2/\mu_i^2}$$

In this case, the arguments are the same as in table 4.1, but in this case, the entries are all named **part**, and the arguments are now the attribute keys:

```
<PotentialSeries active='True'>
  <part potential='gaussian' constant='1000.0' mu_length='0.1' />
  <part potential='gaussian' constant='-60.0' mu_length='0.3' />
  <part potential='exponential' constant='-120.0' mu_length='0.7' />
  <part potential='coulomb' constant='-10.0' mu_length='1.25' />
  <part potential='gaussian_power' constant='-200.0' mu_length='1.75' n_power='2' />
  <part potential='power' constant='-10.0' mu_length='3.5' n_power='3' />
  <part potential='yukawa' constant='2000.0' mu_length='6.5' />
</PotentialSeries>
```

4.5 Brink_Boeker (JT scheme)

Brinl-Boeker ⁷ is a well-known effective potential dependent on the exchange forces, defined by two Gaussians of different lengths.

$$V_{BB} = \sum_{i=1,2} e^{-(r/\mu_i)^2} (V_W^i + V_B^i P^\sigma - V_H^i P^\tau - V_M^i P^\sigma P^\tau) \quad (6)$$

⁷M.Brink, E. Boeker. Effective interactions for Hartree-Fock calculations (1966)

in this case, the arguments are the exchange constants for each gaussian, named as attributes "part_1" and "part_2" for all the parameters. Wigner is the central part, Barlett correspond to the exchange of the spin state, Heisenber to the isospin state and Majorana to the spatial state.

```
<Brink_Boeker>
  <mu_length part_1='0.7'      part_2='1.4'      units='fm' />
  <Wigner     part_1='595.55'   part_2='-72.21'   units='MeV' />
  <Majorana   part_1='-206.05'  part_2='-68.39'   units='MeV' />
  <Bartlett   part_1='0.0'      part_2='0.0'      units='MeV' />
  <Heisenberg part_1='0.0'      part_2='0.0'      units='MeV' />
</Brink_Boeker>
```

4.6 Density_Dependent (JT scheme)

The density-dependent interaction is a contribution of a contact interaction of the two particles with a potential spatially proportional to the whole nucleus in the center-of-mass frame. Since this density depends on the nuclear wave function generated by the matrix elements, this program only gives a Fermi approximation for the particle occupation, which can serve as a seed for further calculations with Taurus.

$$V_{dd} = t_0 (1 + x_0 P^\sigma) \rho^\alpha \left(\frac{\mathbf{r}_1 + \mathbf{r}_2}{2} \right) \delta(\mathbf{r}_1 - \mathbf{r}_2)$$

$$\rho(r) = \langle \Phi | \hat{\rho}(\mathbf{r}) | \Phi \rangle = \langle \Phi | \sum_{ij} \langle i | \delta(r - \hat{r}) | j \rangle c_i^\dagger c_j | \Phi \rangle = \sum_{ij} \langle i | \delta(r - \hat{r}) | j \rangle \langle \Phi | c_i^\dagger c_j | \Phi \rangle$$

where we use the approximation of the occupation of the density matrix as $\langle \Phi | c_i^\dagger c_j | \Phi \rangle = \rho_{ji} = 0$ unless states $\varphi_i, \varphi_j < \epsilon_F$, which lead to

$$\rho(r) = \sum_{i=1}^A |\varphi_i(r)|^2$$

In the implemented approximation, the filling order is strictly symmetrical for protons and neutrons, to reproduce more realistic occupations of medium mass nuclei further development is required.

The arguments are **constant** for the t_0 factor, **alpha** for the dimensionless power of the radial potential, and **x0** for the spin exchange phase factor (default $x_0 = 1$ makes m.e zero for every symmetric S=1 two body wave function). Their value is given for the standard attribute 'value'.

Update (10-2022), for the implementation of density-dependent interactions for no-core valence spaces it is needed a spherical core that could have a different oscillator b-length. A new argument 'core' is required in that case, if no attributes are given (or given empty string "=") the DD term will be computed with A and Z from the SHO parameters. Optional attributes are **protons**, **neutrons** (integers) and **core_b_len** as float (in fm), global b value will be used for empty value given.

```
<Density_Dependent active='True'>
  <constant value='1350.0' units='MeV*fm^-4'>
  <alpha     value='0.33333333' />
  <x0        value='1' />
  <core protons='20' neutrons='20' core_b_len='1.6' />
</Density_Dependent>
```

4.7 Density_From_File (J scheme)

A variant of the previous module to evaluate the Gogny interaction from a general density, imported in terms of the U and V resulting from a HFB calculation. The file must have the format of the TAURUS output for the wave function (in text form, reading the binary is not available for the moment). In order to check the results of TAURUS, it implements the Levedeb+Laguerre integration method to get the interaction in the same form (only the spherical matrix elements).

Introduced/modified tag arguments:

1. **core** is used to know the oscillator length employed in the HFB calculation. Now, this argument has nothing to do with a real inert core (since it does not appear in TAURUS result), then, the particle numbers z/n are ignored for this class. If b is not provided, values are extracted from `<SHO_Parameters>` tag.
2. **file** with attribute **name=** is required to import the wave function.
3. **integration** with integer arguments **r_dim** and **omega_ord** is an optional argument to establish the precision of the integration method (to match with TAURUS calculation). If not provided, the default precision is $R_{dim} = \Omega_{order} = 20$.

```
<Density_From_File active='True'>
  <constant value='1390.6' units='MeV*fm^-4' />
  <alpha value='0.3333' />
  <x0 value='1' />
  <core protons='2' neutrons='2' core_b_len='1.8' />
  <file name='final4HeMz1_b18_wf.txt' />
  <integration r_dim='22' omega_ord='14' />
</Density_From_File>
```

4.8 DensityFiniteRange (JT scheme)

Parameters for Finite range density defined for spherical symmetry and using the Fermi filling approximation, or by importing the wave function with spherical symmetry.

$$V_{frdd} = V(\mathbf{r}_1 - \mathbf{r}_2) (W + BP^\sigma - HP^\tau - MP^\sigma P^\tau) \rho^\alpha \left(\frac{\mathbf{r}_1 + \mathbf{r}_2}{2} \right) \quad (7)$$

Where the relative part of the interaction has the same form as **Central**, using the **n_power** **mu_length** and the **potential**.

Constant α is by default 1/3 but settable. The other arguments combine the options from both previous Density interactions and all optional. But the process for the setting goes in this order:

- 1 **file** require the file to exist and discard the Fermi filling approximation. Only **.txt** is accepted, with Taurus export format*(see below).
- 2 If not present the nucleus will be filled equally with the Fermi approximation. However, the radial integration grid precision can be set.

- 3 For both cases, an internal core can be implemented, with proton/neutron number and oscillator length settable as in the fixed case.

This is an example of the Taurus wave-function format for the U/V s, the order of each sp-state is as it is internally in that program:

```

sp = 1      sh = 1 (0s1/2) mj = +1/2 proton (mt = -1/2)
sp = 2      sh = 1 (0s1/2) mj = -1/2 proton (mt = -1/2)
sp = 3      sh = 103 (0p3/2) mj = +3/2 proton (mt = -1/2)
sp = 4      sh = 103 (0p3/2) mj = +1/2 proton (mt = -1/2)
sp = 5      sh = 103 (0p3/2) mj = -1/2 proton (mt = -1/2)
...
sp = (dimsp)/2      sh = 10101 (1p1/2) mj = -1/2 proton (mt = -1/2)
sp = (dimsp)/2 + 1    sh = 1 (0s1/2) mj = +1/2 neutron (mt = +1/2)
sp = (dimsp)/2 + 2    sh = 1 (0s1/2) mj = -1/2 neutron (mt = +1/2)
... idem

```

From the wave function file:

```

10          ## number of sp-shell states.
1          ## state 1 in Antoine format and l>=10 format
103
101
205
10001
203
307
10103
305
10101
265079390982354432  ## last state label hash (not used here)
0.14522963410101586  ## U [ 0, 1]
0.00000000000000000  ## U [ 0, 2]
0.00000000000000000  ## U [ 0, 3]
...
0.00000000000000000  ## U [ sp_dim, sp_dim]
0.00000000000000000  ## V [ 0, 1]
0.00000000000000000  ## V [ 0, 2]
...

```

Example of situation. Gaussian with exchange factors for the relative coordinate, the *alpha* constant is explicitly set to 1/3, an internal 16 oxygen inert core with an explicit oscillator length of 1.81 fm. For the valence space (defined apart in the Valence space definition) the wave function for the density profile will be set from the file 'final_b198_wf.txt' (that should be done over a valence space). For that COM part of the interaction $\rho^\alpha(\mathbf{R})$, the integral will be radial (Associated-Laguerre) for 22 r grid points.


```

<DensityFiniteRange active='True'>
  <potential name='gaussian'/>
  <Wigner value='134.0' units='MeV'/>
  <Majorana value='10.0' units='MeV'/>
  <Bartlett value='115.0' units='MeV'/>
  <Heisenberg value='10.0' units='MeV'/>
  <mu_length value='1.2' units='fm'/>
  <n_power value='0'/>
  <alpha value='0.3333'/>
  <core protons='8' neutrons='8' core_b_len='1.81'/>
  <file name='final_b198_wf.txt' />
  <integration r_dim='22' omega_ord='14'/>
</DensityFiniteRange>

```

4.9 Kinetic_2Body (JT scheme)

The kinetic energy two-body matrix element is a term required to reproduce the binding energy in no-core calculations, corresponding to a center of mass correction. This matrix element is also non-parametrizable.

```

<Kinetic_2Body active='True'>
</Kinetic_2Body>

```

Kinetic term can produce 1-body matrix elements (hamil_type=3).

4.10 SpinOrbitShortRange (JT scheme)

This is the long wave limit of the analytic spin-orbit interaction and is required for the Gogny D1S force.

$$V_{LS}(\mathbf{r}_1, \mathbf{r}_2) \approx i \frac{W_{LS}}{4} (\sigma_1 + \sigma_2) \cdot (\nabla'_1 - \nabla'_2) \times (\nabla_1 - \nabla_2)$$

Where ∇'_i/∇_i are the differential operator acting on the particle i on the ket/bra. Sigma are the usual Pauli matrices for the particle 1/2.

It takes the same arguments as the Central interaction, but only the constant value is settable internally:

```

<SpinOrbitShortRange active='True'>
  <potential name='power'/>
  <constant value='115.0' units='MeV*fm^-5'/>
  <mu_length value='1.0' units='fm'/>
  <n_power value='0'/>
</SpinOrbitShortRange>

```

4.11 SpinOrbit and Tensor (JT scheme)

These methods implement an analytical resolution of the two-body matrix elements for the SHO wave functions using Racah algebra, by Moshinsky⁸. Using the Talmi method, the radial potential can be extended to the same forms of the Central potential.

$$V_{LS}(\mathbf{r}) = V_{LS}(r) \mathbf{l} \cdot \mathbf{S}$$

Being the relative angular momentum and the total spin.

$$\mathbf{l} = (\mathbf{r}_1 - \mathbf{r}_2) \times (\mathbf{p}_1 - \mathbf{p}_2) \quad \mathbf{S} = \frac{\sigma_1 + \sigma_2}{2}$$

The tensor potential is expressed as a scalar product of a range 2 spherical harmonic tensor and another 2-range spin cross-product tensor.

$$V_T(\mathbf{r}) = \sqrt{\frac{24\pi}{5}} V_T(r) \mathbf{Y}^{(2)}(\hat{\mathbf{r}}) \cdot (\sigma_1 \times \sigma_2)^{(2)}$$

```
<SpinOrbit active='True'>
  <potential name='power' />
  <constant value='1.0' units='MeV' />
  <mu_length value='1.0' units='fm' />
  <n_power value='0' />
</SpinOrbit>

<Tensor active='True'>
  <potential name='power' />
  <constant value='1.0' units='MeV' />
  <mu_length value='1.0' units='fm' />
  <n_power value='0' />
</Tensor>
```

4.11.1 Tensor S_{12} interaction with Exchange operators: TensorS12 (JT scheme).

Based on the previous tensor operator for any central interaction, we generalize the interaction to include the exchange operators defined as in the **Brink-Boeker** interaction (be aware of the sign convention).

$$V_T = V_T(r)(V_W + V_B P^\sigma - V_H P^\tau - V_M P^\sigma P^\tau) \quad (8)$$

This interaction employs both the parameters for the Central type interaction, with optional exchange constants (parameter **constant** is ignored by the program, if given and no exchange operators present, its value will be assigned to the central term.)

```
<TensorS12 active='True'>
  <potential name='gaussian' />
  <Wigner value='-135.0' units='MeV' />
  <Majorana value='0.0' units='MeV' />
  <Barlett value='0.0' units='MeV' />
  <Heisenberg value='-115.0' units='MeV' />
  <mu_length value='1.2' units='fm' />
</TensorS12>
```

⁸T. Brody, M. Moshinsky Tables of transformation Brackets for nuclear shell model calculations (1960) and M. Moshinsky Y. Smirnoff The harmonic oscillator in modern physics (1996)

4.11.2 Spin-Orbit finite range with Exchange operators: SpinOrbitFiniteRange, SpinOrbitSquaredFiniteRange and OrbitalMomentumSquared

The same concerns of the update of the Central term and the TensorS12.

The definitions of the Exchange operators are as in equation (8), being the angular operators SpinOrbitFiniteRange for $(\mathbf{l} \cdot \mathbf{S})$, SpinOrbitSquaredFiniteRange for $(\mathbf{l} \cdot \mathbf{S})^2$, and OrbitalMomentumSquared as \mathbf{l}^2 :

```
<SpinOrbitFiniteRange active='True'>
  <potential name='gaussian' />
  <Wigner value='134.0' units='MeV' />
  <Majorana value='10.0' units='MeV' />
  <Bartlett value='115.0' units='MeV' />
  <Heisenberg value='10.0' units='MeV' />
  <mu_length value='1.2' units='fm' />
  <n_power value='0' />
</SpinOrbitFiniteRange>

<SpinOrbitSquaredFiniteRange active='True'>
  ... </SpinOrbitSquaredFiniteRange>

<OrbitalMomentumSquared active='True'>
  ... </OrbitalMomentumSquared>
```

4.12 SDI / MSDI Surface Delta Interaction (JT scheme)

Zero-range and isospin-dependant. The modified version appends a repulsive constant for the diagonal matrix elements in order to correct the binding energy misbehavior of the SDI (See Brussaard & Glaudemans book (1977)):

$$V_{SDI}(r) = -\frac{V_T}{4\pi} \delta(\vec{r}_1 - \vec{r}_2) \delta(r_1 - R_0)$$

$$V_{MSDI}(r) = V_{SDI}(r) + B\tau(1) \cdot \tau(2) + C$$

Where R_0 is the average nuclear radius. From internal considerations of the HOF at this point, the value of the interaction does not change with the wave functions, so the radial part is approximated by two constants $A_{T=0}$ $A_{T=1}$. The arguments follow the name format from the book.

```
<SDI active='True'>
  <constants AT0='1' AT1='1' B='1.0' C='1.0' units='Mev' />
</SDI>
```

4.13 Multipole Delta Interaction (JT scheme)

Zero-range and isospin-dependant. Multipole expansion from Suhonen's book (pg 213, 8.40-47).

$$v^{spin-isospin}(|\mathbf{r}_1 - \mathbf{r}_2|) = V(|\mathbf{r}_1 - \mathbf{r}_2|) (A + B\sigma_1 \cdot \sigma_2 + C\tau_1 \cdot \tau_2 + D(\sigma_1 \cdot \sigma_2)(\tau_1 \cdot \tau_2))$$

$$V(|\mathbf{r}_1 - \mathbf{r}_2|) = \frac{\delta(r_1 - r_2)}{r_1 r_2} \sum_{\lambda} \mathbf{Y}_{\lambda}(\Omega_1) \cdot \mathbf{Y}_{\lambda}(\Omega_2)$$

The Delta class is an implementation of the abstract `CentralForces.Multipole_JTScheme`, this is an abstract class where the method for the radial integral must be implemented to use another interaction (separable or not) `Multipole_JTScheme.RadialCoeff(self, L)`. Radial integral is defined as

$$\mathcal{R}_{abcd}^{\lambda} = \int_0^{\infty} r_1^2 dr_1 \int_0^{\infty} r_2^2 dr_2 \phi_{n_a l_a}(r_1) \phi_{n_b l_b}(r_2) v_{(\lambda)}(r_1, r_2) \phi_{n_c l_c}(r_1) \phi_{n_d l_d}(r_2)$$

Set class-attribute `SEPARABLE_MULTIPOLE=False` if the radial 2-body integral depends on the exchange (otherwise $\mathcal{R}_{abdc}^{\lambda} = \mathcal{R}_{abcd}^{\lambda}$). Default parameters are $A = 1$ $B = C = D = 0$ as SDI.

```
<Multipole_Delta active='True'>
  <constants A='1.0' B='0.26' C='-1' D='-2.36' units='MeV' />
</Multipole_Delta>
```

4.14 Multipole_Moment Interaction (JT scheme)

The multipole operators are combinations of the spherical harmonic expansion of an interaction, i.e. the Quarupole-Quadrupole interaction. The expression of the interaction is

$$V_{\lambda} = C_{\lambda} r^{\lambda} (\mathbf{Y}_{\lambda} \cdot \mathbf{Y}_{\lambda}) \quad (9)$$

```
<Multipole_Moment active='True'>
  <constant value='0.30' units='MeV*fm^-n_power' />
  <n_power value='2' />
</Multipole_Moment>
```

4.15 SkyrmeBulk_JTScheme Interaction (JT scheme)

Zero-range and isospin-dependant. Evaluation of the first part of the Skyrme interaction as LS matrix elements. Definition from Ring Suck book (pg 175):

$$t_0 (1 + x_0 P^{\sigma}) \delta(\mathbf{r}_{12}) + t_1 \left(\frac{\delta(\mathbf{r}_{12}) |\mathbf{k}|^2 + |\mathbf{k}'|^2 \delta(\mathbf{r}_{12})}{2} \right) + t_2 \mathbf{k}' \delta(\mathbf{r}_{12}) \cdot \mathbf{k}$$

where $\mathbf{k} = \frac{1}{2i}(\nabla_1 - \nabla_2)$ acts on the ket and $\mathbf{k}' = \mathbf{k}^*$ on the bra

Work in progress

```
<SkyrmeBulk active='False'>
  <t0 value='-1128.75' units='MeV*fm^3' />
  <x0 value='0.45' />
  <t1 value='395.0' units='MeV*fm^5' />
  <t2 value='-95.0' units='MeV*fm^3' />
</SkyrmeBulk>
```

4.16 Force_From_File

This interaction gets a whole previous calculation and appends directly to the interaction. The valence space of the previous interaction could be greater than the new one, it will only take the matrix elements with single-particle wave functions in the new valence space (i.e, if you import from S+P+SD shells and now you are calculating S+P shell, it will omit all crossing shell m.e between S+P and SD.). The function reorders the matrix elements for the s.p states to be increasing from bra to ket, multiplying the corresponding phase of the permutation.

It imports as *J-scheme* from '.2b' files and as *JT-scheme* from '.sho' (Hamil.type 1), the arguments are:

1. **file** with attribute **name** to set the path of the interaction.
2. The optional arguments to import in the tag **options**:
 - (a) **ignorelines** (integer) lines to skip the title, the function only reads the JT blocks.
 - (b) **constant** (float) If constant is given, the imported values will be multiplied.
 - (c) **l_ge_10** ('True' or 'False') if the matrix elements are explicitly counting orbital *l* in the lower than 10 format. The default is 'True'.

Be careful because:

1. It cannot know the $\hbar\omega$ or *b* you use in the previous calculation.
2. You can import from a greater valence space but if you import from a smaller you are getting erroneously all new crossing matrix elements as zero.
3. This importing procedure doesn't care about parity conservation, angular momentum addition, etc. Further upgrades can prompt errors if these errors appear.
4. Use the '.2b' or '.sho' only in the matrix element file extension. It cannot determine the scheme without them (for the moment).

```
<Force_From_File active='True'>
  <!-- Import an interaction already saved -->
  <file name='results/D1S_t0_SPSPDPF.2b' />
  <options ignorelines='4' constant='1.0' l_ge_10='True' />
</Force_From_File>
```

5 Debugging and Decomposition of Matrix elements (XLog class)

There are two main debuggers, a simple text saver called `Log` and a tree logger called **XLog** (since saves in XML format). Matrix elements come from nested series, in these cases, the execution on a level makes a step when all the iterations in its subroutines are finished. Tree datatype is suitable for this case, I use XML.etree library to manage it and save it as .xml file.

The nesting might be cumbersome, so debugging is recommended for individual matrix elements. The logger saves information by using the static method `XLog.write(tag, **attributes)`, where the attributes are the details and variables we want in the log. The rules for the tags to be open (relative to an arbitrary node) are:

1. If the tag is not in the branch, it is added (as well as its attributes).
2. Else, verifies the value of the attributes given. Attribute labels are free for the name, but it is recommended to be given once.
 - (a) If new attributes are in the tag, it will append them to the current node (if old keys are given, their values will be overwritten). This allows us to write in the same tag in different lines (i.e., after the coefficients from intermediate subroutines are obtained).
 - (b) When attributes are all in the current node, it's taken as a new iteration, the previous branch is closed, and another one is created. Closed branches cannot be reopened, if new attributes match with another branch ones it will separately save them (tree nodes work as lists).

Note that the order of logging is crucial because the logger does not distinguish between the program levels. If you want to obtain the value of a nested process (with other logs in it), you must open the tag before the call and then write again with the result, otherwise, the result of that level will appear at the end of the branch.

Node creation should be done in the same method, calling from other inner methods is not recommended (but works as well).

Let's see an example, Coulomb matrix element is evaluated in the following order: 1) Normalization and antisymmetrization (evaluates the m.e. again with the exchanged ket), 2) Recoupling from J to LS scheme, 3) Evaluation of the center of mass transformation (Talmi integral series).

First, we insert the comments with the condition of a debugging state. Then we save different values, f.e. in the step 1) we do:

```
...
1  if self.DEBUG_MODE:
2      XLog.write('na_me', p='DIRECT', ket=self.ket.shellStatesNotation)
3
4  direct = self._non_antisymmetrized_ME()
5
6  if self.DEBUG_MODE:
7      XLog.write('na_me', value=direct)
8      XLog.write('na_me', p='EXCHANGED', ket=exchanged_ket.shellStatesNotation)
```

```

9
10 exchan = exch_2bme._non_antisymmetrized_ME()
...
16 if self.DEBUG_MODE:
17     XLog.write('na_me', value=exchan, phs=phase)
18     XLog.write('nas', norms=(self.bra.norm(), self.ket.norm()), value=self._value)
...

```

In this level, we have defined a 'nas' tag, with the name of the states in an upper method. Then we will evaluate the non-antisymmetrized matrix element of the direct element and the exchanged element, I want to save both branches, so I save a common node 'na_me' but with a different label 'p'. This lets the *direct* branch be nested in the first place and append the value of the resulting matrix element before closing it and doing the same for the exchanged m.e. The line 18, proves that previous tags can be updated from inner methods (not recommended). The process is repeated similarly in the LS decomposition('recoup') and the Talmi series ('talmi') methods, it prompts the following result for the matrix element $\langle (0p_{1/2})^2 | V_{Coul} | (0p_{1/2})^2 \rangle_{J=0, pp}$

```

<?xml version="1.0" ?>
<root>
  <nas bra="0p1/2 0p1/2" ket="0p1/2 0p1/2" norms="(0.7071, 0.7071)" value="0.8690">
    <na_me p="DIRECT" ket="0p1/2 0p1/2" value="0.8690">
      <recoup Lb="0" Sb="0" val_b="0.5774" Lk="0" Sk="0" val_k="0.5774">
        <talmi p="0" series="1.2500" Ip="0.7979" val="0.9974"/>
        <talmi p="1" series="-1.5000" Ip="0.5319" val="-0.7979"/>
        <talmi p="2" series="1.2500" Ip="0.4255" val="0.5319"/>
      </recoup>
      <recoup Lb="1" Sb="1" val_b="0.8165" Lk="1" Sk="1" val_k="0.8165">
        <talmi p="0" series="0.0" Ip="0.7979" val="0.0"/>
        <talmi p="1" series="1.0000" Ip="0.5319" val="0.5319"/>
        <talmi p="2" series="0.0" Ip="0.4255" val="0.0"/>
      </recoup>
    </na_me>
    <na_me p="EXCHANGED" ket="0p1/2 0p1/2" value="0.8690" phs="-1">
      <recoup Lb="0" Sb="0" val_b="0.5774" Lk="0" Sk="0" val_k="0.5774">
        <talmi p="0" series="1.2500" Ip="0.7979" val="0.9974"/>
        <talmi p="1" series="-1.5000" Ip="0.5319" val="-0.7979"/>
        <talmi p="2" series="1.2500" Ip="0.4255" val="0.5319"/>
      </recoup>
      <recoup Lb="1" Sb="1" val_b="0.8165" Lk="1" Sk="1" val_k="0.8165">
        <talmi p="0" series="0.0" Ip="0.7979" val="0.0"/>
        <talmi p="1" series="1.0000" Ip="0.5319" val="0.5319"/>
        <talmi p="2" series="0.0" Ip="0.4255" val="0.0"/>
      </recoup>
    </na_me>
  </nas>
</root>

```

XML editor plugins can help us collapse the branches.

6 Comparing matrix elements

6.1 Comparing two files in JT scheme for testing. `MatrixElementFilesComparator`

With this helper (`helpers.MatrixElementFilesComparator`), we give two files (in the same scheme and Antoine format) in the JT scheme. Given a bench file, it compares them by notice:

1. Telling apart between diagonal $\langle ab|A|ab\rangle$ and off diagonal matrix elements $\langle ab|A|cd\rangle$ when counting the missings/failures.
2. The quantum numbers exchange, it adds the phase in case of rearranging. It doesn't leave matrix elements if do not appear in the same order or form.
3. Counting how many of the "test" elements are missing in the "bench" and how many are missing in from the "bench" that appear in the "test" file.

Usage:

1. Instance the class naming the files (1st is the bench, 2nd is the one to compare)
`MatrixElementFilesComparator(file_bench, file_test)`, there are other optional arguments:
 - (a) `ignorelines` is a tuple with the title lines to omit at the beginning of the file, default is `=(4,4)`
 - (b) `l_ge_10=True` is the default Antoine format to read as mentioned before, both files have to have the same.
 - (c) `verbose` prompts in console all the process of comparison, by default is disabled.
2. Evaluate the comparison calling the method `compareDictionaries()`
3. Get the different results with: `getResults()` (for a count of the states by diagonal/off-diagonal) and the specific fails with `getFailedME()`, `getMissingME()`. Notice that Failed matrix elements are considered in the intersection of matrix element blocks since it specify the relative/absolute difference between them. Elements outside the intersection are labeled as miss and skipped.

Let's see for example two identical sets of central matrix elements obtained, where we change the matrix element $\langle 0s_{1/2}, 0s_{1/2}|V|0s_{1/2}, 0s_{1/2}\rangle_{1,0}$ by a sign, then we remove the matrix element block $\langle 0s_{1/2}, 0s_{1/2}|V|0p_{1/2}, 0p_{1/2}\rangle$ from the test file and the $\langle 0s_{1/2}, 0s_{1/2}|V|1s_{1/2}, 1s_{1/2}\rangle$ from the bench one:⁹

⁹`helpers.prettyPrintDictionary()` is an auxiliary method to print in the console the dictionaries by nesting them in vertical instead of horizontal.


```

test = MatrixElementFilesComparator('../results/central_SPSD_bench.sho',
                                     '../results/central_SPSD_2.sho',
                                     # optional arguments
                                     ignorelines=(4,4), l_ge_10=True, verbose=False)
_result = test.compareDictionaries()

print(" === TEST RESULTS:      =====\n")
prettyPrintDictionary(test.getResults())
>>>
TOTAL:428
FAIL:3
PASS:425
DIAGONAL: {
  . ZERO: {
    . . MISS:0
    . . TOTAL:67
    . . OK:67
  . }
  . NON_ZERO: {
    . . MISS:0
    . . TOTAL:45
    . . OK:44
  . }
}
OFF_DIAG: {
  . ZERO: {
    . . MISS:0
    . . TOTAL:198
    . . OK:198
  . }
  . NON_ZERO: {
    . . MISS:2
    . . TOTAL:118
    . . OK:116
  . }
}
<<<
prettyPrintDictionary(test.getFailedME())
>>>
(1, 1, 1, 1): {
  . JT:1,0: {
    . . abs:[5.367925]t != [-5.367925]b
    . . rel:[-1.0000=-100.000%]
  . }
}
<<<
prettyPrintDictionary(test.getMissingME())
>>>
in_bench:[(1, 1, 10001, 10001)]
in_test:[(1, 1, 101, 101)]
<<<

```

6.2 Comparing a (reduced valence space) sets of matrix elements by plotting their relative differences.

Developing.