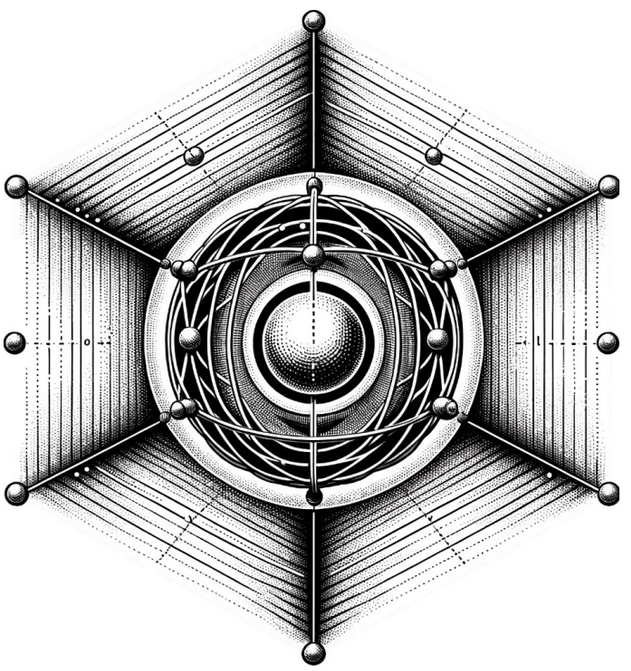
qlanth

v1.4.1

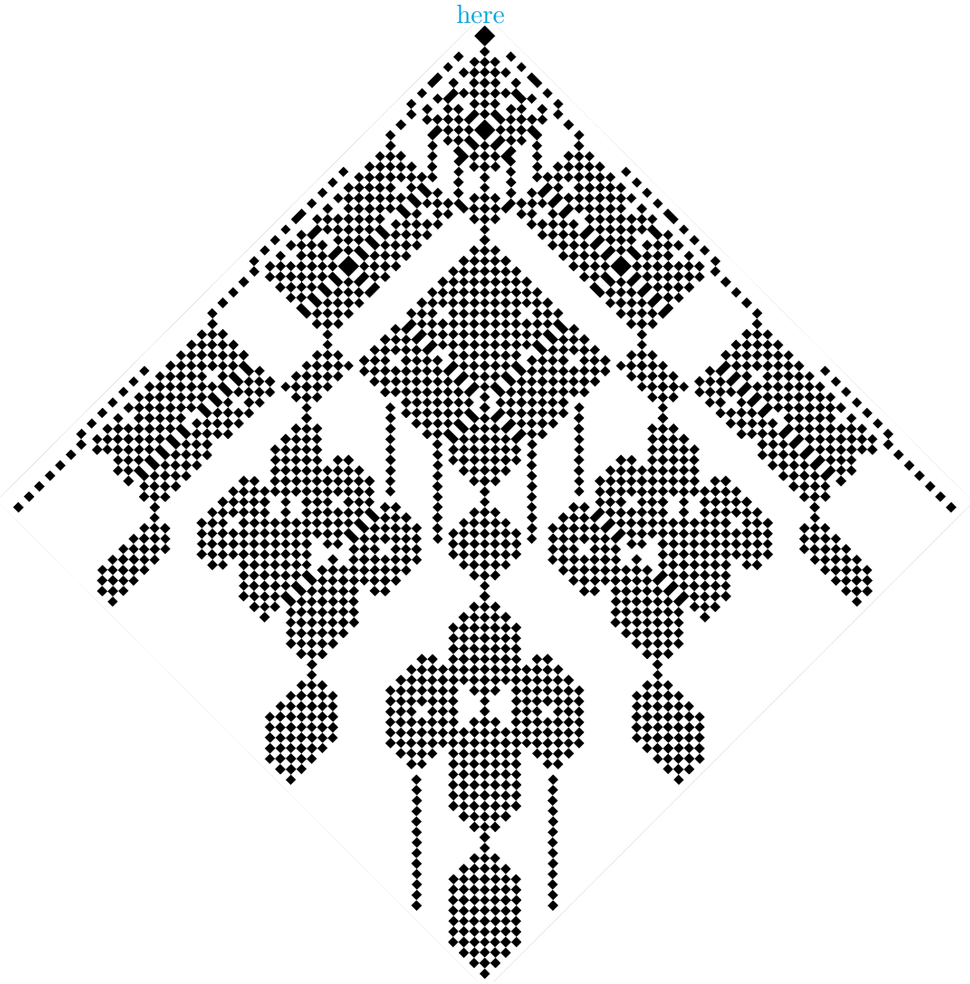


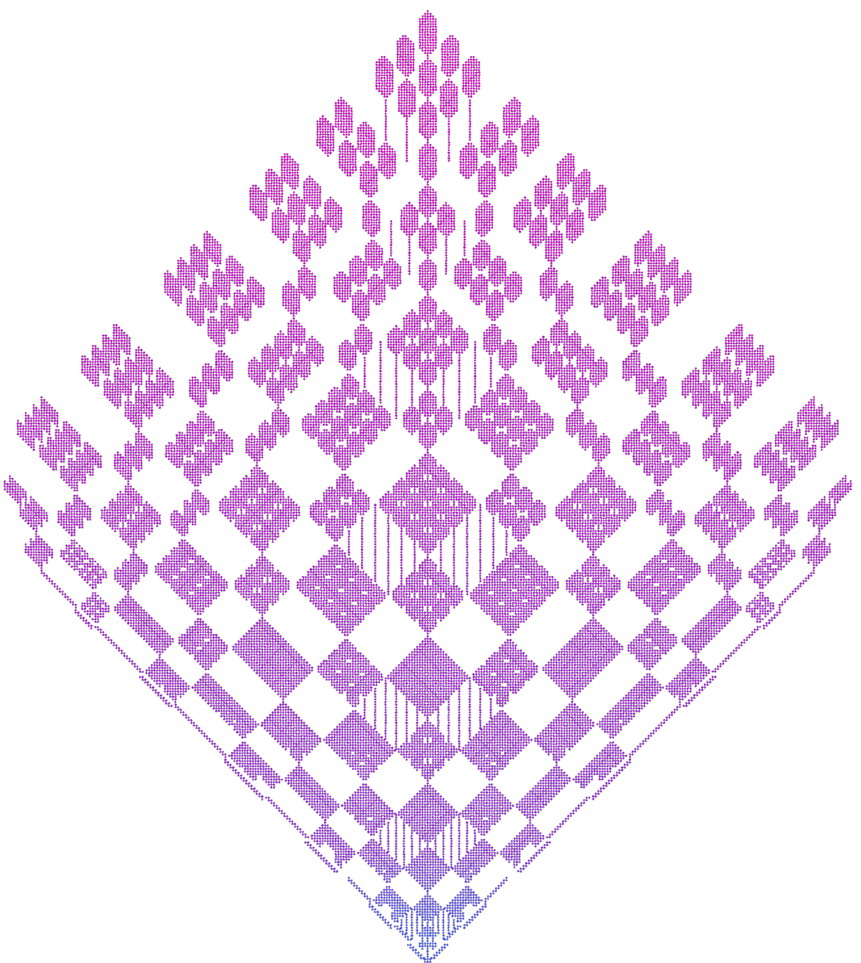
Juan David Lizarazo Ferro,
Christopher Dodson
& Rashid Zia

Brown University,
Department of Physics

Providence, Rhode Island
2025 AD

qlanth may be downloaded
here





This work was sponsored by the
National Science Foundation
Grant No. DMR-1922025

qlanth is a tool that can be used to estimate the electronic structure of lanthanide ions in crystals. I uses an effective Hamiltonian limited to a single-configuration with included configuration-interaction corrections. This Hamiltonian aims to describe the observed properties of ions embedded in solids in a picture that imagines them as free-ions modified by the influence of the lattice in which they find themselves in.

This picture of lanthanide ions is one that developed and mostly matured in the second half of the last century by the efforts of John Slater,[1] Giulio Racah,[2] Brian Judd,[3] Gerhard Dieke,[4] Hannah Crosswhite,[5] Robert Cowan,[6] Michael Reid,[7] William Carnall,[8] Clyde Morrison,[9] Richard Leavitt,[10] Brian Wybourne,[11] Richard Trees,[12] and Katherine Rajnak[13] among others. The goal of this tool is to provide a modern implementation of the methods that resulted from their work. This code is written in Wolfram language.

Separate to their specific use in this code, **qlanth** also includes data that might be of use to those interested in the single-configuration description of lanthanide ions. These data include the coefficients of fractional parentage (as calculated by Velkov and parsed here), and reduced matrix elements for all the operators in the effective Hamiltonian. These are provided as standard *Mathematica* associations that should be simple to use elsewhere. One feature of **qlanth** is that symbolic expressions are maintained up to the very last moment where numerical approximations are inevitable. As such, the symbolic expressions that result for the matrix representation of the Hamiltonian, result in linear combinations of the model parameters with symbolic coefficients.

The included *Mathematica* notebook `qlanth.nb` lists most of the functions included in **qlanth** and should be considered complementary to this document. The `/examples` folder includes notebooks containing the result of this description for most of the trivalent lanthanide ions in lanthanum fluoride. LaF_3 is remarkable in that it was one of the systems in which a systematic study [Car+89] of all of the trivalent lanthanide ions were studied.

This code was originally authored by Christopher Dodson and Rashid Zia for their research into magnetic dipole transitions in lanthanide ions [DZ12]. Here it has been rewritten and expanded by David Lizarazo. It has also benefited from conversations with Tharnier Puel at the University of Iowa.

This document has 18 sections. Section 1 gives an overview the semi-empirical Hamiltonian. Section 2 explains the details of the basis in which the semi-empirical Hamiltonian is evaluated, together with the method of fractional parentage, additional quantum numbers, Kramer's degeneracy, and the JJ' block structure of the semi-empirical Hamiltonian. Section 3 gives a detailed explanation of each of the interactions include in the semi-empirical Hamiltonian. Section 4 gives explains the implicit assumptions in the orientation of the coordinate system. Section 5 gives an overview of the attendant experimental setups and considerations about uncertainty. Section 6 is about the calculation of magnetic and forced electric dipole transitions. Section 7 explain certain constraints often used for the parameters in the semi-empirical Hamiltonian.

Section 8 explains the details of fitting the Hamiltonian to experimental data.Section 9 lists included auxiliary *Mathematica* notebooks. Section 11 explains the details of an abbreviated Python extension to **qlanth**. Section 12 explains some of the included experimental data. Section 13 contains a few assorted details on running **qlanth**. Section 14 has a brief comment on units. Section 15 and Section 17 include a summary of notation and definitions used throughout this document. Finally, Section 18 contains a printout of the code included in **qlanth**.

Besides being a fully functional code that works out of the box, **qlanth** is unique in that it also includes computational routines that can generate from scratch (or close to scratch) the necessary reduced matrix elements which in other codes are simply loaded from other vintages. Great care was taken to comment every loop, variable, procedure, and data provenance. To highlight this, the code relevant to the different functions has been interspersed in the parts where they are mentioned.

[1] [Sla29] [2] [Rac42a; Rac42b; Rac43; Rac49] [3] [Jud62; Jud63b; Jud63a; Jud66; Jud67; JCC68; CCJ68; Jud82; Jud83; JS84; JC84; Jud85; Jud86; Jud88; Jud89; JL93; Jud96; Jud05] [4] [DC63; PDC67; Die68] [5] [CCJ68; Cro71; Cro+76; Cro+77; DC63; JCC68; JC84] [6] [Cow81] [7] [Rei81] [8] [CFW65; Car+89; Car92; CFR68b; CFR68e; CFR68c; CFR68d; CFR68a; Car+70; Car+76; GW+91] [9] [MWK76; ML79; MW94; Mor80; MT87; MKW77b; MKW77a; ML82; Mor+83] [10] [Lea87; Lea82; LM80; ML79; ML82] [11] [CFW65; CW63; RW63; RW64b; RW64a; Wyb64a; Wyb64b; Wyb65; Wyb70; WS07] [12] [Tre52; Tre51; Tre58] [13] [RW63; RW64b; RW64a; Raj65]

Contents

1 The semi-empirical Hamiltonian

Electrons in a multi-electron ion are subject to a number of interactions. They are attracted to the nucleus about which they orbit. Being bundled together with other electrons, they experience repulsion from all of them. Having spin, they are also subject to various magnetic interactions. The spin of each electron interacts with the magnetic field generated by its own orbital angular momentum and of other electrons. And between pairs of electrons, the spin of one can influence the others spin through the interaction of their respective magnetic dipoles.

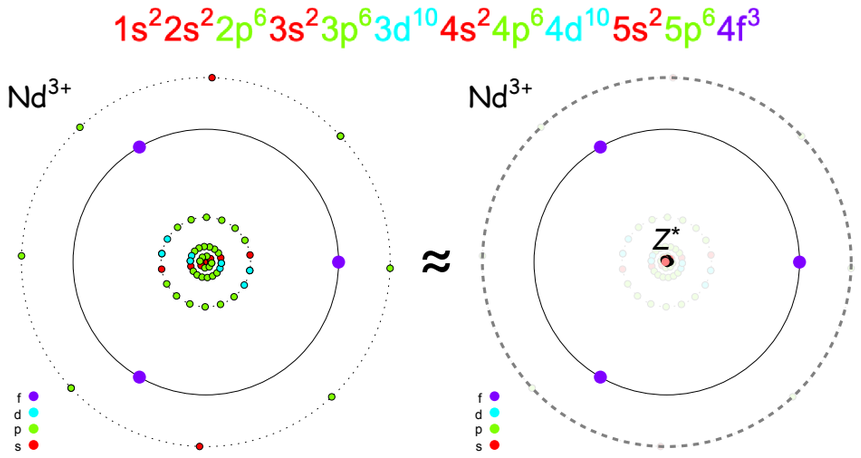


Figure 1: The trivalent neodymium ion shielded by $5s^2$ and $5p^6$ closed shells.

To describe the effect of the charges in the lattice surrounding the ion, the crystal field is introduced. In the simplest of embodiments, the crystal field is simply seen as the electrostatic field due to surrounding charges. This model is of limited applicability if taken too literally; however, if only symmetry considerations are assumed, the model is seen to have greater validity but a somewhat less clear physical origin.

The Hilbert space of a multi-electron ion is a vast stage. In principle, a basis for it should have a countable infinity of bound states and an uncountable infinity of unbound states. This is clearly too much to handle, but thankfully, this large stage can be put in some order thanks to the exclusion principle. The exclusion principle (together with that graceful tendency of things to drift downwards the energetic wells) provides the shell structure. This shell structure, in turn, makes it possible that an atom with many electrons, can be described effectively as an aggregate of an inert core, and a fewer active valence electrons.

Take for instance a triply ionized (or trivalent) neodymium atom, as depicted in Fig-1. In principle, this gives us the daunting task of dealing with the enormous Hilbert space of 57 electrons. However, 54 of them arrange themselves in a xenon core, so that we are only left to deal with only three. Three are still a challenging task, but much less so than 57. Furthermore, the exclusion principle also guides us in what type of orbital we could possibly place these three electrons, in the case of the lanthanide ions, this being the 4f orbitals. But not really, there are many more unoccupied orbitals outside of the xenon core, two of these electrons, if they are willing to pay the energetic price, they could find themselves in a 5d or a 6s orbital.

Here we shall assume a single-configuration description. Meaning that all the valence electrons in the ions that we study will all be considered to be located in f-orbitals, or what is the same, that they are described by $\underline{f}^n$ wavefunctions. Table 2 shows the (ground) configuration for the trivalent lanthanide ions. This is, however, a harsh approximation, but thankfully one can make some corrections to it. The effects that arise in the single configuration description because of omitting all the other possible orbitals where the

58	59	60	61	62	63	64	65	66	67	68	69	70
Ce^{3+}	Pr^{3+}	Nd^{3+}	Pm^{3+}	Sm^{3+}	Eu^{3+}	Gd^{3+}	Tb^{3+}	Dy^{3+}	Ho^{3+}	Er^{3+}	Tm^{3+}	Yb^{3+}
[Xe] f^1	[Xe] f^2	[Xe] f^3	[Xe] f^4	[Xe] f^5	[Xe] f^6	[Xe] f^7	[Xe] f^8	[Xe] f^9	[Xe] f^{10}	[Xe] f^{11}	[Xe] f^{12}	[Xe] f^{13}
Cerium	Praseodymium	Neodymium	Promethium	Samarium	Europium	Gadolinium	Terbium	Dysprosium	Holmium	Erbium	Thulium	Ytterbium

Figure 2: The trivalent lanthanide row and their ground configurations.

electrons might find themselves, this is what is called *configuration-interaction*.

These effects can be brought within the simplified description through perturbation theory. The task not the usual one of correcting for the energies/eigenvectors given an added perturbation, but rather to consider the effects of using a truncated Hilbert space due to a known interaction. For a detailed analysis of this, see Rudzikas' book [Rud07] on theoretical atomic spectroscopy or this article [Lin74] by Lindgren. What results from this analysis are operators that now act solely within the single configuration but with a coefficient that depends on overlap integrals between different configurations. It is from *configuration-interaction* that the parameters $\alpha, \beta, \gamma, P^{(k)}, T^{(k)}$ enter into the description.

$$\hat{\mathcal{H}} = \underbrace{\hat{\mathcal{H}}_{\text{k}}}_{\text{kinetic}} + \underbrace{\hat{\mathcal{H}}_{\text{e:sn}}}_{\text{e:shielded nuc}} + \overbrace{\underbrace{\hat{\mathcal{H}}_{\text{s:o}}}_{\text{spin-orbit}} + \underbrace{\hat{\mathcal{H}}_{\text{s:s}}}_{\substack{\text{spin:spin} \\ \text{and spin:other-orbit}}} + \underbrace{\hat{\mathcal{H}}_{\text{s:oo⊕ecs:o}}}_{\substack{\text{spin:other-orbit} \\ \text{ec-correlated-spin:orbit}}}}^{\text{magnetic interactions}} + \underbrace{\hat{\mathcal{H}}_{\text{Z}}}_{\text{Zeeman}}$$

$$+ \overbrace{\underbrace{\hat{\mathcal{H}}_{\text{e:e}}}_{\text{e:e}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{G}_2}}_{\text{G}_2\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(7)}}_{\mathcal{SO}(7)\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\text{⋏}}}_{\substack{\text{effective} \\ \text{three-body}}} + \underbrace{\hat{\mathcal{H}}_{\text{cf}}}_{\text{crystal field}}}^{\text{electric interactions}} \tag{1}$$

$$\hat{\mathcal{H}}_{\text{k}} = -\frac{\hbar^2}{2m_e}\sum_{i=1}^{n}\nabla_i^2 \text{ (kinetic energy of } n \text{ valence electrons)} \tag{2}$$

$$\hat{\mathcal{H}}_{\text{e:sn}} = \sum_{i=1}^{n} V_{\text{sn}}(r_i) \text{ (valence-electrons interaction with shielded nuc. charge)} \tag{3}$$

$$\hat{\mathcal{H}}_{\text{s:o}} = \begin{cases} \sum_{i=1}^{n}\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) & \text{with } \xi(r_i) = \frac{\hbar^2}{2m^2c^2r_i}\frac{\mathrm{d}V_{\text{sn}}(r_i)}{\mathrm{d}r_i} \\ \sum_{i=1}^{n}\zeta\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) & \substack{\text{with } \zeta \text{ the radial average of } \xi(r_i) \\ \text{or used as phenomenological parameter}} \end{cases} \tag{4}$$

$$\hat{\mathcal{H}}_{\text{s:s}} = \sum_{k=0,2,4} m^{(k)}\hat{m}_k{}^{ss} \tag{5}$$

$$\hat{\mathcal{H}}_{\text{s:oo⊕ecs:o}} = \sum_{k=2,4,6} P^{(k)}\hat{p}_k + \sum_{k=0,2,4} m^{(k)}\hat{m}_k \tag{6}$$

$$\hat{\mathcal{H}}_{\text{Z}} = -\vec{B}\cdot\hat{\mu} = \mu_B\vec{B}\cdot\left(\hat{L} + g_s\hat{S}\right) \text{ (interaction with a magnetic field)} \tag{7}$$

$$\hat{\mathcal{H}}_{\text{e:e}} = \sum_{i>j}^{n,n}\frac{e^2}{\|\vec{r}_i - \vec{r}_j\|} = \sum_{k=0,2,4,6} F^{(k)}\hat{f}_k \text{ (repulsion between valence electrons)} \tag{8}$$

Let $\hat{\mathcal{C}}(\mathcal{G}) \coloneqq$ The Casimir operator of group $\mathcal{G}$.

$$\hat{\mathcal{H}}_{\mathcal{SO}(3)} = \alpha\,\hat{\mathcal{C}}(\mathcal{SO}(3)) = \alpha\hat{L}^2 \text{ (Trees effective operator)} \tag{9}$$

$$\hat{\mathcal{H}}_{\mathcal{G}_2} = \beta\,\hat{\mathcal{C}}(\mathcal{G}_2) \tag{10}$$

$$\hat{\mathcal{H}}_{\mathcal{SO}(7)} = \gamma\,\hat{\mathcal{C}}(\mathcal{SO}(7)) \tag{11}$$

$$\hat{\mathcal{H}}_{\text{⋏}} = T'^{(2)}\hat{t}_2{}' + T'^{(11)}\hat{t}_{11}{}' + \sum_{\substack{k=2,3,4,6,7,8,\\ 11,12,14,15,\\ 16,17,18,19}} T^{(k)}\hat{t}_k \text{ (effective 3-body int.)} \tag{12}$$

$$\hat{\mathcal{H}}_{\text{cf}} = \sum_{i=1}^{n} V_{\text{CF}}(\hat{r}_i) = \sum_{i=1}^{n}\sum_{k=2,4,6}\sum_{q=-k}^{k}\mathcal{B}_q^{(k)}C_q^{(k)}(i) \quad \substack{\text{(crystal field interaction} \\ \text{with surroundings)}} \tag{13}$$

One could try to evaluate the coefficients that result in the Hamiltonian. However, within the **semi-empirical** approach, these parameters are left to be fitted against experimental data, or at times approximated through Hartree-Fock analysis. This approach is only *semi* empirical in the sense that the model parameters are fitted from experimental data, but the semi-empirical Hamiltonian that is fitted is based on a clear physical picture inherited from atomic physics.

Putting all of this together leads to the following effective Hamiltonian as show in &qn-1, where "v-electrons" is shorthand for valence electrons. It is important to note that the eigenstates that we'll end up with have shoved under the rug all the radial dependence of the wavefunctions. This dependence having been integrated in the parameters of the effective Hamiltonian. The resulting wavefunctions being solely concerned with the angular dependence of the wavefunctions, but modulated by the effects of the radial dependence.

Once all the parameters in this semi-empirical Hamiltonian have been fitted to experimental data what results is a Hamiltonian such as the one for Pr^{3+} in LaF_3 shown

in 𝓕ig-3. Before we go on to explain in some detail each of the terms included in this Hamiltonian, let us continue to explain the basis used in calculations.

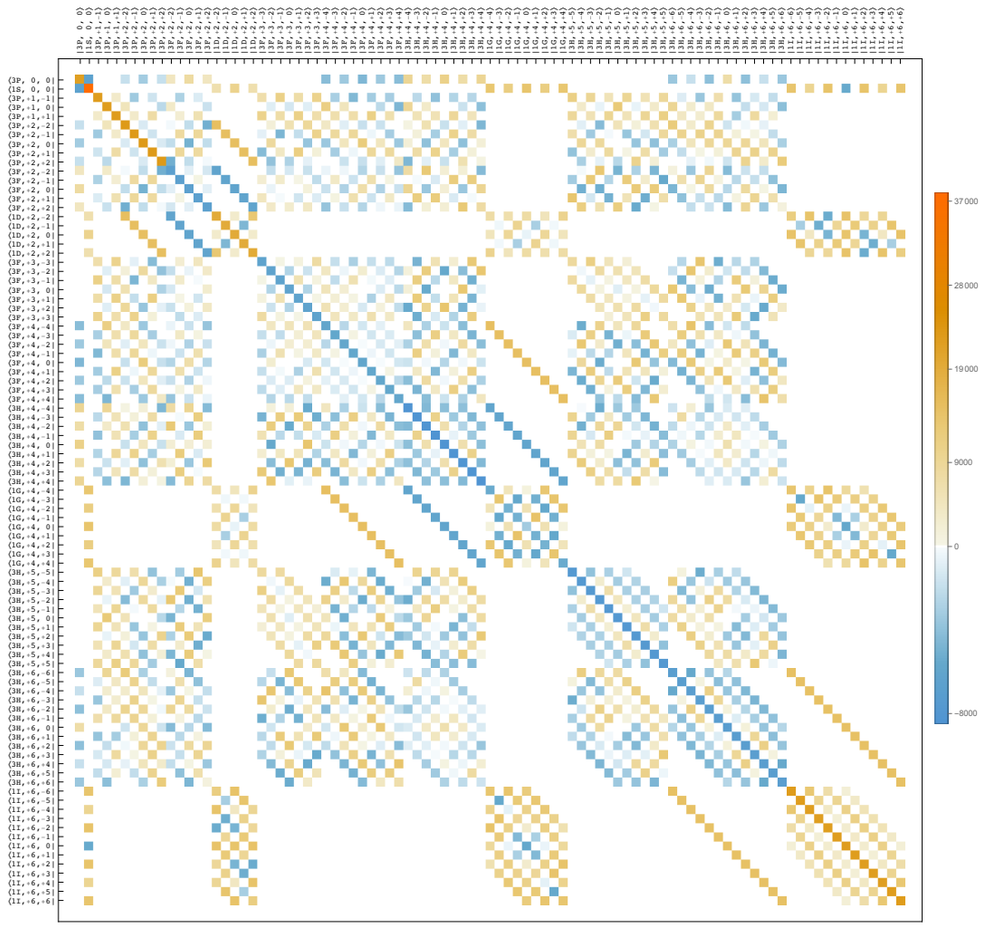


Figure 3: The matrix representation of $\hat{\mathcal{H}}$ for Pr^{3+} in LaF_3 in the $|LSJM_J\rangle$ basis.

2 LS coupling basis

In choosing a coupling scheme (or equivalently, choosing a basis in which to represent the Hamiltonian), there are a myriad options; all of them legitimate in their own right. The art of choosing a useful coupling scheme is that of proposing a basis for the angular part of the wavefunctions that will be close to the actual eigenstates of the system. It being necessary to calculate the matrix elements of the relevant operators, choosing a coupling scheme may also be justified by the easy by which these can be calculated.

`qlanth` uses LS coupling for its calculations. In LS coupling all the orbital angular momenta are added to form the total orbital angular momentum L, all the spin angular momenta are added to form the total spin angular momentum S, and finally these two angular momenta are then added together to form the total angular momentum J. The exclusion principle is taken into account in limiting the possible LS terms, and demands no further restrictions. Finally this total angular momentum J is complemented with the quantum number[14] M_J describing the projection of J along the z-axis.

It is worthwhile remembering here the spectroscopic hierarchy of descriptive elements: **terms** correspond to $|LS\rangle$ (also noted as $^{2S+1}\mathrm{L}$), **levels** correspond to $|LSJ\rangle$ (also noted as $^{2S+1}\mathrm{L}_J$), and **states** correspond to $|LSJM_J\rangle$ (also noted as $^{2S+1}\mathrm{L}_{J,M_J}$). 𝓕ig-4 shows an example of the relationship between a term and its associated levels and states.

In principle the $|LSJM_J\rangle$ description is the primordial one, the $|LSJ\rangle$ resulting from neglecting all parts of the Hamiltonian that have no spherical symmetry, and the $|LS\rangle$ resulting from further neglecting all terms that couple the spin and orbital angular momenta. Note that a *state* is not an *eigen*-state; all of these are assumed to be basis vectors in the type of description attached to them.

[14] A *good* quantum number is any eigenvalue of an operator that commutes with the Hamiltonian; in other words, they are conserved quantities.

Whereas all four quantum numbers $|LSJM_J\rangle$ are required to specify a state, one may, however, use two simpler descriptions as the situation merits. When all the parts of the Hamiltonian without spherical symmetry are excluded, then a description in terms of $|LSJ\rangle$ levels is sufficient, the M_J quantum numbers being redundant and with J being a good quantum number. In a second scenario, when in addition to neglecting all parts without spherical symmetry, one also neglects all parts of the Hamiltonian that couple the spin and orbital degrees of freedom, then the $|LS\rangle$ terms constitute the most parsimonious description, with L and S being separately conserved quantities.

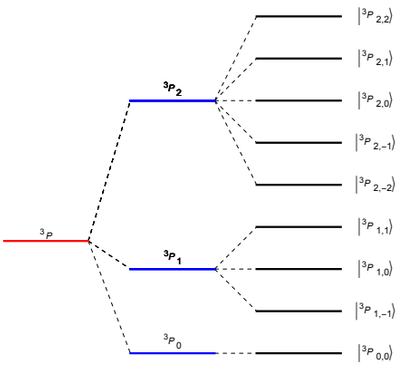


Figure 4: Levels and states associated with the ^3P term in $\underline{\mathrm{f}}^2$.

When a certain level of description has been adopted one can then assume (at one's own peril) that single states, levels, or terms are actual *eigen*-states/levels/terms of the system at hand. This assumption results in simple transition rules between states/levels/terms. One may, however, within each level of description, take an alternate route, the *intermediate coupling* route, of seeing how the different states/levels/terms mix in the eigenstates found by diagonalizing the appropriate Hamiltonian. This results in a more detailed description at the cost of increased complexity.

2.1 $|LSJM_J\rangle$ states

The basis vectors of the $|LSJM_J\rangle$ basis are common eigenvectors of the operators $\hat{L}^2$, $\hat{S}^2$, $\hat{J}^2$, and $\hat{J}_z$. They are formed starting from the allowed LS terms in a given configuration, and are then completed with attendant J and M_J quantum numbers. The LS terms allowed in each configuration $\underline{\mathrm{f}}^n$ are obtained from tables that originate from the original work by Nielson and Koster [NK63]. In **qlanth** these terms are parsed from the file `B1F_-ALL.TXT` which is part of the doctoral research of Dobromir Velkov (under the advisory of Brian Judd) [Vel00] in which he calculated anew the coefficients of fractional parentage.

One of the facts that have to be accounted for in a basis that uses L and S as quantum numbers, is that there might be several linearly independent paths to couple the electron spin and orbital momenta to add up to given total L and total S. For this reason additional labels are necessary to distinguish between these different terms. The simplest way of doing this dates back to the tables of Nielson and Koster [NK63], and consists in assigning consecutive integers to degenerate LS terms, with no further meaning to these integers, except that of discriminating between degenerate terms.

The following are all the LS terms in the $\underline{\mathrm{f}}^n$ configurations. In the notation used, the superscript index before the letter notes the spin multiplicity $2S+1$, the roman letter indicates the value of L in spectroscopic notation ($S\to1, P\to2, D\to3, F\to4, G\to5, H\to6, I\to7, K\to8, L\to9, M\to10, N\to11, O\to12, Q\to3, R\to14, T\to15, U\to16, V\to17$), and the final integer (if present) is the label that discriminates between several degenerate LS and must not be confused with a value of J. This last index we frequently label in the equations contained in this document with the greek letter α (sadly, for historical reasons, we prepend it, rather than append it).

$\underline{\mathrm{f}}^0$
(1 LS term)
^{1}S

$\underline{\mathrm{f}}^1$
(1 LS term)
^{2}F

$\underline{\mathrm{f}}^2$
(7 LS terms)

^{3}P, ^{3}F, ^{3}H, ^{1}S, ^{1}D, ^{1}G, ^{1}I

$\underline{\underline{f}}^3$
(17 LS terms)

^{4}S, ^{4}D, ^{4}F, ^{4}G, ^{4}I, ^{2}P, ^{2}D1, ^{2}D2, ^{2}F1, ^{2}F2, ^{2}G1, ^{2}G2, ^{2}H1, ^{2}H2, ^{2}I, ^{2}K, ^{2}L

$\underline{\underline{f}}^4$
(47 LS terms)

^{5}S, ^{5}D, ^{5}F, ^{5}G, ^{5}I, ^{3}P1, ^{3}P2, ^{3}P3, ^{3}D1, ^{3}D2, ^{3}F1, ^{3}F2, ^{3}F3, ^{3}F4, ^{3}G1, ^{3}G2, ^{3}G3, ^{3}H1, ^{3}H2, ^{3}H3, ^{3}H4, ^{3}I1, ^{3}I2, ^{3}K1, ^{3}K2, ^{3}L, ^{3}M, ^{1}S1, ^{1}S2, ^{1}D1, ^{1}D2, ^{1}D3, ^{1}D4, ^{1}F, ^{1}G1, ^{1}G2, ^{1}G3, ^{1}G4, ^{1}H1, ^{1}H2, ^{1}I1, ^{1}I2, ^{1}I3, ^{1}K, ^{1}L1, ^{1}L2, ^{1}N

$\underline{\underline{f}}^5$
(73 LS terms)

^{6}P, ^{6}F, ^{6}H, ^{4}S, ^{4}P1, ^{4}P2, ^{4}D1, ^{4}D2, ^{4}D3, ^{4}F1, ^{4}F2, ^{4}F3, ^{4}F4, ^{4}G1, ^{4}G2, ^{4}G3, ^{4}G4, ^{4}H1, ^{4}H2, ^{4}H3, ^{4}I1, ^{4}I2, ^{4}I3, ^{4}K1, ^{4}K2, ^{4}L, ^{4}M, ^{2}P1, ^{2}P2, ^{2}P3, ^{2}P4, ^{2}D1, ^{2}D2, ^{2}D3, ^{2}D4, ^{2}D5, ^{2}F1, ^{2}F2, ^{2}F3, ^{2}F4, ^{2}F5, ^{2}F6, ^{2}F7, ^{2}G1, ^{2}G2, ^{2}G3, ^{2}G4, ^{2}G5, ^{2}G6, ^{2}H1, ^{2}H2, ^{2}H3, ^{2}H4, ^{2}H5, ^{2}H6, ^{2}H7, ^{2}I1, ^{2}I2, ^{2}I3, ^{2}I4, ^{2}I5, ^{2}K1, ^{2}K2, ^{2}K3, ^{2}K4, ^{2}K5, ^{2}L1, ^{2}L2, ^{2}L3, ^{2}M1, ^{2}M2, ^{2}N, ^{2}O

$\underline{\underline{f}}^6$
(119 LS terms)

^{7}F, ^{5}S, ^{5}P, ^{5}D1, ^{5}D2, ^{5}D3, ^{5}F1, ^{5}F2, ^{5}G1, ^{5}G2, ^{5}G3, ^{5}H1, ^{5}H2, ^{5}I1, ^{5}I2, ^{5}K, ^{5}L, ^{3}P1, ^{3}P2, ^{3}P3, ^{3}P4, ^{3}P5, ^{3}P6, ^{3}D1, ^{3}D2, ^{3}D3, ^{3}D4, ^{3}D5, ^{3}F1, ^{3}F2, ^{3}F3, ^{3}F4, ^{3}F5, ^{3}F6, ^{3}F7, ^{3}F8, ^{3}F9, ^{3}G1, ^{3}G2, ^{3}G3, ^{3}G4, ^{3}G5, ^{3}G6, ^{3}G7, ^{3}H1, ^{3}H2, ^{3}H3, ^{3}H4, ^{3}H5, ^{3}H6, ^{3}H7, ^{3}H8, ^{3}H9, ^{3}I1, ^{3}I2, ^{3}I3, ^{3}I4, ^{3}I5, ^{3}I6, ^{3}K1, ^{3}K2, ^{3}K3, ^{3}K4, ^{3}K5, ^{3}K6, ^{3}L1, ^{3}L2, ^{3}L3, ^{3}M1, ^{3}M2, ^{3}M3, ^{3}N, ^{3}O, ^{1}S1, ^{1}S2, ^{1}S3, ^{1}S4, ^{1}P, ^{1}D1, ^{1}D2, ^{1}D3, ^{1}D4, ^{1}D5, ^{1}D6, ^{1}F1, ^{1}F2, ^{1}F3, ^{1}F4, ^{1}G1, ^{1}G2, ^{1}G3, ^{1}G4, ^{1}G5, ^{1}G6, ^{1}G7, ^{1}G8, ^{1}H1, ^{1}H2, ^{1}H3, ^{1}H4, ^{1}I1, ^{1}I2, ^{1}I3, ^{1}I4, ^{1}I5, ^{1}I6, ^{1}I7, ^{1}K1, ^{1}K2, ^{1}K3, ^{1}L1, ^{1}L2, ^{1}L3, ^{1}L4, ^{1}M1, ^{1}M2, ^{1}N1, ^{1}N2, ^{1}Q

$\underline{\underline{f}}^7$
(119 LS terms)

^{8}S, ^{6}P, ^{6}D, ^{6}F, ^{6}G, ^{6}H, ^{6}I, ^{4}S1, ^{4}S2, ^{4}P1, ^{4}P2, ^{4}D1, ^{4}D2, ^{4}D3, ^{4}D4, ^{4}D5, ^{4}D6, ^{4}F1, ^{4}F2, ^{4}F3, ^{4}F4, ^{4}F5, ^{4}G1, ^{4}G2, ^{4}G3, ^{4}G4, ^{4}G5, ^{4}G6, ^{4}G7, ^{4}H1, ^{4}H2, ^{4}H3, ^{4}H4, ^{4}H5, ^{4}I1, ^{4}I2, ^{4}I3, ^{4}I4, ^{4}I5, ^{4}K1, ^{4}K2, ^{4}K3, ^{4}L1, ^{4}L2, ^{4}L3, ^{4}M, ^{4}N, ^{2}S1, ^{2}S2, ^{2}P1, ^{2}P2, ^{2}P3, ^{2}P4, ^{2}P5, ^{2}D1, ^{2}D2, ^{2}D3, ^{2}D4, ^{2}D5, ^{2}D6, ^{2}D7, ^{2}F1, ^{2}F2, ^{2}F3, ^{2}F4, ^{2}F5, ^{2}F6, ^{2}F7, ^{2}F8, ^{2}F9, ^{2}F10, ^{2}G1, ^{2}G2, ^{2}G3, ^{2}G4, ^{2}G5, ^{2}G6, ^{2}G7, ^{2}G8, ^{2}G9, ^{2}G10, ^{2}H1, ^{2}H2, ^{2}H3, ^{2}H4, ^{2}H5, ^{2}H6, ^{2}H7, ^{2}H8, ^{2}H9, ^{2}I1, ^{2}I2, ^{2}I3, ^{2}I4, ^{2}I5, ^{2}I6, ^{2}I7, ^{2}I8, ^{2}I9, ^{2}K1, ^{2}K2, ^{2}K3, ^{2}K4, ^{2}K5, ^{2}K6, ^{2}K7, ^{2}L1, ^{2}L2, ^{2}L3, ^{2}L4, ^{2}L5, ^{2}M1, ^{2}M2, ^{2}M3, ^{2}M4, ^{2}N1, ^{2}N2, ^{2}O, ^{2}Q

$\underline{\underline{f}}^8$
(119 LS terms)

^{7}F, ^{5}S, ^{5}P, ^{5}D1, ^{5}D2, ^{5}D3, ^{5}F1, ^{5}F2, ^{5}G1, ^{5}G2, ^{5}G3, ^{5}H1, ^{5}H2, ^{5}I1, ^{5}I2, ^{5}K, ^{5}L, ^{3}P1, ^{3}P2, ^{3}P3, ^{3}P4, ^{3}P5, ^{3}P6, ^{3}D1, ^{3}D2, ^{3}D3, ^{3}D4, ^{3}D5, ^{3}F1, ^{3}F2, ^{3}F3, ^{3}F4, ^{3}F5, ^{3}F6, ^{3}F7, ^{3}F8, ^{3}F9, ^{3}G1, ^{3}G2, ^{3}G3, ^{3}G4, ^{3}G5, ^{3}G6, ^{3}G7, ^{3}H1, ^{3}H2, ^{3}H3, ^{3}H4, ^{3}H5, ^{3}H6, ^{3}H7, ^{3}H8, ^{3}H9, ^{3}I1, ^{3}I2, ^{3}I3, ^{3}I4, ^{3}I5, ^{3}I6, ^{3}K1, ^{3}K2, ^{3}K3, ^{3}K4, ^{3}K5, ^{3}K6, ^{3}L1, ^{3}L2, ^{3}L3, ^{3}M1, ^{3}M2, ^{3}M3, ^{3}N, ^{3}O, ^{1}S1, ^{1}S2, ^{1}S3, ^{1}S4, ^{1}P, ^{1}D1, ^{1}D2, ^{1}D3, ^{1}D4, ^{1}D5, ^{1}D6, ^{1}F1, ^{1}F2, ^{1}F3, ^{1}F4, ^{1}G1, ^{1}G2, ^{1}G3, ^{1}G4, ^{1}G5, ^{1}G6, ^{1}G7, ^{1}G8, ^{1}H1, ^{1}H2, ^{1}H3, ^{1}H4, ^{1}I1, ^{1}I2, ^{1}I3, ^{1}I4, ^{1}I5, ^{1}I6, ^{1}I7, ^{1}K1, ^{1}K2, ^{1}K3, ^{1}L1, ^{1}L2, ^{1}L3, ^{1}L4, ^{1}M1, ^{1}M2, ^{1}N1, ^{1}N2, ^{1}Q

$\underline{f}^9$ (73 LS terms)
^{6}P, ^{6}F, ^{6}H, ^{4}S, ^{4}P1, ^{4}P2, ^{4}D1, ^{4}D2, ^{4}D3, ^{4}F1, ^{4}F2, ^{4}F3, ^{4}F4, ^{4}G1, ^{4}G2, ^{4}G3, ^{4}G4, ^{4}H1, ^{4}H2, ^{4}H3, ^{4}I1, ^{4}I2, ^{4}I3, ^{4}K1, ^{4}K2, ^{4}L, ^{4}M, ^{2}P1, ^{2}P2, ^{2}P3, ^{2}P4, ^{2}D1, ^{2}D2, ^{2}D3, ^{2}D4, ^{2}D5, ^{2}F1, ^{2}F2, ^{2}F3, ^{2}F4, ^{2}F5, ^{2}F6, ^{2}F7, ^{2}G1, ^{2}G2, ^{2}G3, ^{2}G4, ^{2}G5, ^{2}G6, ^{2}H1, ^{2}H2, ^{2}H3, ^{2}H4, ^{2}H5, ^{2}H6, ^{2}H7, ^{2}I1, ^{2}I2, ^{2}I3, ^{2}I4, ^{2}I5, ^{2}K1, ^{2}K2, ^{2}K3, ^{2}K4, ^{2}K5, ^{2}L1, ^{2}L2, ^{2}L3, ^{2}M1, ^{2}M2, ^{2}N, ^{2}O

$\underline{f}^{10}$ (47 LS terms)
^{5}S, ^{5}D, ^{5}F, ^{5}G, ^{5}I, ^{3}P1, ^{3}P2, ^{3}P3, ^{3}D1, ^{3}D2, ^{3}F1, ^{3}F2, ^{3}F3, ^{3}F4, ^{3}G1, ^{3}G2, ^{3}G3, ^{3}H1, ^{3}H2, ^{3}H3, ^{3}H4, ^{3}I1, ^{3}I2, ^{3}K1, ^{3}K2, ^{3}L, ^{3}M, ^{1}S1, ^{1}S2, ^{1}D1, ^{1}D2, ^{1}D3, ^{1}D4, ^{1}F, ^{1}G1, ^{1}G2, ^{1}G3, ^{1}G4, ^{1}H1, ^{1}H2, ^{1}I1, ^{1}I2, ^{1}I3, ^{1}K, ^{1}L1, ^{1}L2, ^{1}N

$\underline{f}^{11}$ (17 LS terms)
^{4}S, ^{4}D, ^{4}F, ^{4}G, ^{4}I, ^{2}P, ^{2}D1, ^{2}D2, ^{2}F1, ^{2}F2, ^{2}G1, ^{2}G2, ^{2}H1, ^{2}H2, ^{2}I, ^{2}K, ^{2}L

$\underline{f}^{12}$ (7 LS terms)
^{3}P, ^{3}F, ^{3}H, ^{1}S, ^{1}D, ^{1}G, ^{1}I

$\underline{f}^{13}$ (1 LS term)
^{2}F

$\underline{f}^{14}$ (1 LS term)
^{1}S

In **qlanth** these terms may be queried through the function `AllowedNKSLTerms`.

```
AllowedNKSLTerms::usage = "AllowedNKSLTerms[numE] returns a list with
    the allowed terms in the f^numE configuration, the terms are
   given as strings in spectroscopic notation. The integers in the
   last positions are used to distinguish cases with degeneracy.";
AllowedNKSLTerms[numE_] := Map[First, CFPTerms[Min[numE, 14-numE]]];
AllowedNKSLTerms[0]  = {"1S"};
AllowedNKSLTerms[14] = {"1S"};
```

In addition to LS, the $|LSJM_J\rangle$ basis states are also specified by the total angular momentum J (which may go from $|L-S|$ to $|L+S|$). Then for each J, there are $2J+1$ projections on the z-axis. For example, the ordered $|LSJM_J\rangle$ basis for $\underline{f}^2$ is shown below, where the first element is the LS term given as a string, the second equal to J, and the third one equal to M_J:

$(J=0)$ (2 states)
$\|^3\mathrm{P}_{0,0}\rangle$, $\|^1\mathrm{S}_{0,0}\rangle$

$(J = 1)$
(3 states)

$|^3\mathrm{P}_{1,-1}\rangle$, $|^3\mathrm{P}_{1,0}\rangle$, $|^3\mathrm{P}_{1,1}\rangle$

$(J = 2)$
(15 states)

$|^3\mathrm{P}_{2,-2}\rangle$, $|^3\mathrm{P}_{2,-1}\rangle$, $|^3\mathrm{P}_{2,0}\rangle$, $|^3\mathrm{P}_{2,1}\rangle$, $|^3\mathrm{P}_{2,2}\rangle$, $|^3\mathrm{F}_{2,-2}\rangle$, $|^3\mathrm{F}_{2,-1}\rangle$, $|^3\mathrm{F}_{2,0}\rangle$, $|^3\mathrm{F}_{2,1}\rangle$, $|^3\mathrm{F}_{2,2}\rangle$, $|^1\mathrm{D}_{2,-2}\rangle$, $|^1\mathrm{D}_{2,-1}\rangle$, $|^1\mathrm{D}_{2,0}\rangle$, $|^1\mathrm{D}_{2,1}\rangle$, $|^1\mathrm{D}_{2,2}\rangle$

$(J = 3)$
(7 states)

$|^3\mathrm{F}_{3,-3}\rangle$, $|^3\mathrm{F}_{3,-2}\rangle$, $|^3\mathrm{F}_{3,-1}\rangle$, $|^3\mathrm{F}_{3,0}\rangle$, $|^3\mathrm{F}_{3,1}\rangle$, $|^3\mathrm{F}_{3,2}\rangle$, $|^3\mathrm{F}_{3,3}\rangle$

$(J = 4)$
(27 states)

$|^3\mathrm{F}_{4,-4}\rangle$, $|^3\mathrm{F}_{4,-3}\rangle$, $|^3\mathrm{F}_{4,-2}\rangle$, $|^3\mathrm{F}_{4,-1}\rangle$, $|^3\mathrm{F}_{4,0}\rangle$, $|^3\mathrm{F}_{4,1}\rangle$, $|^3\mathrm{F}_{4,2}\rangle$, $|^3\mathrm{F}_{4,3}\rangle$, $|^3\mathrm{F}_{4,4}\rangle$, $|^3\mathrm{H}_{4,-4}\rangle$, $|^3\mathrm{H}_{4,-3}\rangle$, $|^3\mathrm{H}_{4,-2}\rangle$, $|^3\mathrm{H}_{4,-1}\rangle$, $|^3\mathrm{H}_{4,0}\rangle$, $|^3\mathrm{H}_{4,1}\rangle$, $|^3\mathrm{H}_{4,2}\rangle$, $|^3\mathrm{H}_{4,3}\rangle$, $|^3\mathrm{H}_{4,4}\rangle$, $|^1\mathrm{G}_{4,-4}\rangle$, $|^1\mathrm{G}_{4,-3}\rangle$, $|^1\mathrm{G}_{4,-2}\rangle$, $|^1\mathrm{G}_{4,-1}\rangle$, $|^1\mathrm{G}_{4,0}\rangle$, $|^1\mathrm{G}_{4,1}\rangle$, $|^1\mathrm{G}_{4,2}\rangle$, $|^1\mathrm{G}_{4,3}\rangle$, $|^1\mathrm{G}_{4,4}\rangle$

$(J = 5)$
(11 states)

$|^3\mathrm{H}_{5,-5}\rangle$, $|^3\mathrm{H}_{5,-4}\rangle$, $|^3\mathrm{H}_{5,-3}\rangle$, $|^3\mathrm{H}_{5,-2}\rangle$, $|^3\mathrm{H}_{5,-1}\rangle$, $|^3\mathrm{H}_{5,0}\rangle$, $|^3\mathrm{H}_{5,1}\rangle$, $|^3\mathrm{H}_{5,2}\rangle$, $|^3\mathrm{H}_{5,3}\rangle$, $|^3\mathrm{H}_{5,4}\rangle$, $|^3\mathrm{H}_{5,5}\rangle$

$(J = 6)$
(26 states)

$|^3\mathrm{H}_{6,-6}\rangle$, $|^3\mathrm{H}_{6,-5}\rangle$, $|^3\mathrm{H}_{6,-4}\rangle$, $|^3\mathrm{H}_{6,-3}\rangle$, $|^3\mathrm{H}_{6,-2}\rangle$, $|^3\mathrm{H}_{6,-1}\rangle$, $|^3\mathrm{H}_{6,0}\rangle$, $|^3\mathrm{H}_{6,1}\rangle$, $|^3\mathrm{H}_{6,2}\rangle$, $|^3\mathrm{H}_{6,3}\rangle$, $|^3\mathrm{H}_{6,4}\rangle$, $|^3\mathrm{H}_{6,5}\rangle$, $|^3\mathrm{H}_{6,6}\rangle$, $|^1\mathrm{I}_{6,-6}\rangle$, $|^1\mathrm{I}_{6,-5}\rangle$, $|^1\mathrm{I}_{6,-4}\rangle$, $|^1\mathrm{I}_{6,-3}\rangle$, $|^1\mathrm{I}_{6,-2}\rangle$, $|^1\mathrm{I}_{6,-1}\rangle$, $|^1\mathrm{I}_{6,0}\rangle$, $|^1\mathrm{I}_{6,1}\rangle$, $|^1\mathrm{I}_{6,2}\rangle$, $|^1\mathrm{I}_{6,3}\rangle$, $|^1\mathrm{I}_{6,4}\rangle$, $|^1\mathrm{I}_{6,5}\rangle$, $|^1\mathrm{I}_{6,6}\rangle$

The order above is an example of the bases ordering used in **qlanth**. Notice how the basis vectors are sorted in order of increasing J, so that for instance not all of the basis states associated with the ^{3}P LS term are contiguous. Within each group for a given J the basis kets are then ordered in decreasing S, then ordered in increasing L, and then according to M_J.

In **qlanth** the ordered basis used for a given $\underline{\mathrm{f}}^n$ is provided by `BasisLSJMJ` which provides a list with $\binom{14}{n}$ elements.

```
BasisLSJMJ::usage = "BasisLSJMJ[numE] returns the ordered basis in L-S-J-MJ with the total orbital angular momentum L and total spin angular momentum S coupled together to form J. The function returns a list with each element representing the quantum numbers for each basis vector. Each element is of the form {SL (string in spectroscopic notation),J, MJ}.
The option ''AsAssociation'' can be set to True to return the basis as an association with the keys corresponding to values of J and the values lists with the corresponding {L, S, J, MJ} list. The default of this option is False.
";
Options[BasisLSJMJ] = {"AsAssociation" -> False};
BasisLSJMJ[numE_, OptionsPattern[]] := Module[
  {energyStatesTable, basis, idx1},
  (
    energyStatesTable = BasisTableGenerator[numE];
    basis = Table[
      energyStatesTable[{numE, AllowedJ[numE][[idx1]]}],
```

```
      {idx1, 1, Length[AllowedJ[numE]]}];
    basis = Flatten[basis, 1];
    If[OptionValue["AsAssociation"],
      (
        Js    = AllowedJ[numE];
        basis = Table[(J -> Select[basis, #[[2]] == J &]), {J, Js}];
        basis = Association[basis];
      )
    ];
    Return[basis];
  )
];
```

2.2 More quantum numbers

Besides using an integer which solves the problem of discriminating between degenerate LS terms by enumerating them, it is also possible to add more useful labels that reflect additional symmetries that the f-electron basis states have in the groups $\mathcal{SO}(7)$ (the Lie group of rotations in seven dimensions) and $\mathcal{G}_2$ (the rank-2 exceptional simple Lie group).

2.2.1 Seniority ν

The seniority number connects different LS terms between configurations, so that a term below can be seen as the *senior* of a term above. To determine the seniority of a given term in configuration $\underline{\mathrm{f}}^n$, one must first find the configuration $\underline{\mathrm{f}}^{\tilde{n}}$ in which this term appeared. For example, $\underline{\mathrm{f}}^5$ contains six degenerate ${}^2G$ terms. The first time this term appeared was in $\underline{\mathrm{f}}^3$, where it had a degeneracy of 2. The 2 degenerate terms in $\underline{\mathrm{f}}^3$ would then both have a seniority of $\boldsymbol{\nu} = 3$ since they first appeared in $\underline{\mathrm{f}}^3$. In consequence two of the six degenerate terms in $\underline{\mathrm{f}}^5$ would have the same degeneracy those two in $\underline{\mathrm{f}}^3$, and are therefore linked to those previous two. The four remaining ones, are considered to be *born* in $\underline{\mathrm{f}}^5$, and therefore have a seniority $\boldsymbol{\nu} = 5$.

These rules seem to be ad-hoc, but they are useful in dealing with the degeneracies in the LS terms as they arrive going up the configurations. It provides a useful way of tracking what happens to each *branch* of the coupling tree as it grows and withers with increasing number of electrons.

There is, however, a deeper meaning to the seniority number. It can be shown that the seniority number (more exactly a quantity related to it) is a sort of spin, a *quasi*-spin, where the spin projections along the 'z-axis' correspond to different number of electrons in $\underline{\mathrm{f}}^n$ configurations [Jud67]. This is a consequence of the exclusion principle. It is also useful to relate matrix elements of operators in one configuration to those in another, through the use of the Wigner-Eckart theorem. This is an interesting and useful theoretical construct, but the method of fractional parentage (which is what is implemented in **qlanth**) is equally adequate, albeit being somewhat less parsimonious than what the quasi-spin view that seniority can provide. As such **qlanth** does not use the seniority numbers that are associated with each LS term[15]. However, in **qlanth** the seniority of a given LS term can be obtained using the function `Seniority`.

```
Seniority::usage = "Seniority[LS] returns the seniority of the given
    term.";
Seniority[LS_] := FindNKLSTerm[LS][[1, 2]];
```

2.2.2 $\mathcal{U}$ and $\mathcal{W}$

Much as L tells us how a rotation acts on an L wavefunction by mixing different M_L components, these other two quantum numbers specify how the wavefunctions transform under the operations of two other two groups. The $\mathcal{W}$ label determines how a wavefunction transforms under a rotation in 7-dimensional space, and $\mathcal{U}$ how they transform under an operator of group $\mathcal{G}_2$. Without going into the group theoretical details, the irreducible representations of $\mathcal{SO}(7)$ can be represented by triples of integer numbers, and those of $\mathcal{G}_2$ as pairs of two integers.

In **qlanth** the $\mathcal{W}$ and $\mathcal{U}$ are used in order to determine the matrix elements of the $\hat{\mathcal{C}}(\mathcal{SO}(7))$ and $\hat{\mathcal{C}}(\mathcal{G}_2)$ Casimir operators. These labels can be retrieved, for a given LS string, using the function `FindNKLSTerm`.

[15] Except for calculating the coefficients of fractional parentage beyond $\underline{\mathrm{f}}^7$, which are useful, but not essential to the calculations of **qlanth**.

```
FindNKLSTerm::usage = "Given the string LS FindNKLSTerm[SL] returns
    all the terms that are compatible with it. This is only for f^n
    configurations. The provided terms might belong to more than one
    configuration. The function returns a list with elements of the
    form {LS, seniority, W, U}.";
FindNKLSTerm[SL_] := Module[
  {NKterms, n},
  (
    n = 7;
    NKterms = {{}};
    Map[
      If[! StringFreeQ[First[#], SL],
        If[ToExpression[Part[#, 2]] <= n,
          NKterms = Join[NKterms, {#}, 1]
        ]
      ] &,
    fnTermLabels
    ];
    NKterms = DeleteCases[NKterms, {}];
    NKterms
  )
];
```

2.3 $|LSJ\rangle$ levels

When the Hamiltonian only includes spherically symmetric terms (or what is the same, when the crystal field is neglected) then the M_J quantum numbers in the $|LSJM_J\rangle$ basis states are redundant. This permits a simplified description in terms of $|LSJ\rangle$ levels. The following are the different $^{2S+1}\mathrm{L}_J$ levels that span the eigenvectors that result from diagonalizing the Hamiltonian in the level description, these may also be termed *multiplets*. (In these we have excluded the indices that distinguish between degenerate LS terms)

$\underline{\mathrm{f}}^1$ (2 LSJ levels)
$^2\mathrm{F}_{5/2}$, $^2\mathrm{F}_{7/2}$

$\underline{\mathrm{f}}^2$ (13 LSJ levels)
$^3\mathrm{P}_0$, $^1\mathrm{S}_0$, $^3\mathrm{P}_1$, $^3\mathrm{P}_2$, $^3\mathrm{F}_2$, $^1\mathrm{D}_2$, $^3\mathrm{F}_3$, $^3\mathrm{F}_4$, $^3\mathrm{H}_4$, $^1\mathrm{G}_4$, $^3\mathrm{H}_5$, $^3\mathrm{H}_6$, $^1\mathrm{I}_6$

$\underline{\mathrm{f}}^3$ (41 LSJ levels)
$^4\mathrm{D}_{1/2}$, $^2\mathrm{P}_{1/2}$, $^4\mathrm{S}_{3/2}$, $^4\mathrm{D}_{3/2}$, $^4\mathrm{F}_{3/2}$, $^2\mathrm{P}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^4\mathrm{D}_{5/2}$, $^4\mathrm{F}_{5/2}$, $^4\mathrm{G}_{5/2}$, $^2\mathrm{D}_{5/2}$, $^2\mathrm{D}_{5/2}$, $^2\mathrm{F}_{5/2}$, $^2\mathrm{F}_{5/2}$, $^4\mathrm{D}_{7/2}$, $^4\mathrm{F}_{7/2}$, $^4\mathrm{G}_{7/2}$, $^2\mathrm{F}_{7/2}$, $^2\mathrm{F}_{7/2}$, $^2\mathrm{G}_{7/2}$, $^2\mathrm{G}_{7/2}$, $^4\mathrm{F}_{9/2}$, $^4\mathrm{G}_{9/2}$, $^4\mathrm{I}_{9/2}$, $^2\mathrm{G}_{9/2}$, $^2\mathrm{G}_{9/2}$, $^2\mathrm{H}_{9/2}$, $^2\mathrm{H}_{9/2}$, $^4\mathrm{G}_{11/2}$, $^4\mathrm{I}_{11/2}$, $^2\mathrm{H}_{11/2}$, $^2\mathrm{H}_{11/2}$, $^2\mathrm{I}_{11/2}$, $^4\mathrm{I}_{13/2}$, $^2\mathrm{I}_{13/2}$, $^2\mathrm{K}_{13/2}$, $^4\mathrm{I}_{15/2}$, $^2\mathrm{K}_{15/2}$, $^2\mathrm{L}_{15/2}$, $^2\mathrm{L}_{17/2}$

$\underline{\mathrm{f}}^4$ (107 LSJ levels)
$^5\mathrm{D}_0$, $^3\mathrm{P}_0$, $^3\mathrm{P}_0$, $^3\mathrm{P}_0$, $^1\mathrm{S}_0$, $^1\mathrm{S}_0$, $^5\mathrm{D}_1$, $^5\mathrm{F}_1$, $^3\mathrm{P}_1$, $^3\mathrm{P}_1$, $^3\mathrm{P}_1$, $^3\mathrm{D}_1$, $^3\mathrm{D}_1$, $^5\mathrm{S}_2$, $^5\mathrm{D}_2$, $^5\mathrm{F}_2$, $^5\mathrm{G}_2$, $^3\mathrm{P}_2$, $^3\mathrm{P}_2$, $^3\mathrm{P}_2$, $^3\mathrm{D}_2$, $^3\mathrm{D}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^5\mathrm{D}_3$, $^5\mathrm{F}_3$, $^5\mathrm{G}_3$, $^3\mathrm{D}_3$, $^3\mathrm{D}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{G}_3$, $^3\mathrm{G}_3$, $^3\mathrm{G}_3$, $^1\mathrm{F}_3$, $^5\mathrm{D}_4$, $^5\mathrm{F}_4$, $^5\mathrm{G}_4$, $^5\mathrm{I}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{G}_4$, $^3\mathrm{G}_4$, $^3\mathrm{G}_4$, $^3\mathrm{H}_4$, $^3\mathrm{H}_4$, $^3\mathrm{H}_4$, $^3\mathrm{H}_4$, $^1\mathrm{G}_4$, $^1\mathrm{G}_4$, $^1\mathrm{G}_4$, $^1\mathrm{G}_4$, $^5\mathrm{F}_5$, $^5\mathrm{G}_5$, $^5\mathrm{I}_5$, $^3\mathrm{G}_5$, $^3\mathrm{G}_5$, $^3\mathrm{G}_5$, $^3\mathrm{H}_5$, $^3\mathrm{H}_5$, $^3\mathrm{H}_5$, $^3\mathrm{H}_5$, $^3\mathrm{I}_5$, $^3\mathrm{I}_5$, $^1\mathrm{H}_5$, $^1\mathrm{H}_5$, $^5\mathrm{G}_6$, $^5\mathrm{I}_6$, $^3\mathrm{H}_6$, $^3\mathrm{H}_6$, $^3\mathrm{H}_6$, $^3\mathrm{H}_6$, $^3\mathrm{I}_6$, $^3\mathrm{I}_6$, $^3\mathrm{K}_6$, $^3\mathrm{K}_6$, $^1\mathrm{I}_6$, $^1\mathrm{I}_6$, $^1\mathrm{I}_6$, $^5\mathrm{I}_7$, $^3\mathrm{I}_7$, $^3\mathrm{I}_7$, $^3\mathrm{K}_7$, $^3\mathrm{K}_7$, $^3\mathrm{L}_7$, $^1\mathrm{K}_7$, $^5\mathrm{I}_8$, $^3\mathrm{K}_8$, $^3\mathrm{K}_8$, $^3\mathrm{L}_8$, $^3\mathrm{M}_8$, $^1\mathrm{L}_8$, $^1\mathrm{L}_8$, $^3\mathrm{L}_9$, $^3\mathrm{M}_9$, $^3\mathrm{M}_{10}$, $^1\mathrm{N}_{10}$

$\underline{\mathrm{f}}^5$ (198 LSJ levels)
$^6\mathrm{F}_{1/2}$, $^4\mathrm{P}_{1/2}$, $^4\mathrm{P}_{1/2}$, $^4\mathrm{D}_{1/2}$, $^4\mathrm{D}_{1/2}$, $^4\mathrm{D}_{1/2}$, $^2\mathrm{P}_{1/2}$, $^2\mathrm{P}_{1/2}$, $^2\mathrm{P}_{1/2}$, $^2\mathrm{P}_{1/2}$, $^6\mathrm{P}_{3/2}$, $^6\mathrm{F}_{3/2}$, $^4\mathrm{S}_{3/2}$, $^4\mathrm{P}_{3/2}$, $^4\mathrm{P}_{3/2}$, $^4\mathrm{D}_{3/2}$, $^4\mathrm{D}_{3/2}$, $^4\mathrm{D}_{3/2}$, $^4\mathrm{F}_{3/2}$, $^4\mathrm{F}_{3/2}$, $^4\mathrm{F}_{3/2}$, $^4\mathrm{F}_{3/2}$, $^2\mathrm{P}_{3/2}$, $^2\mathrm{P}_{3/2}$, $^2\mathrm{P}_{3/2}$, $^2\mathrm{P}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^2\mathrm{D}_{3/2}$, $^6\mathrm{P}_{5/2}$, $^6\mathrm{F}_{5/2}$, $^6\mathrm{H}_{5/2}$, $^4\mathrm{P}_{5/2}$, $^4\mathrm{P}_{5/2}$, $^4\mathrm{D}_{5/2}$, $^4\mathrm{D}_{5/2}$, $^4\mathrm{D}_{5/2}$, $^4\mathrm{F}_{5/2}$, $^4\mathrm{F}_{5/2}$, $^4\mathrm{F}_{5/2}$, $^4\mathrm{F}_{5/2}$, $^4\mathrm{G}_{5/2}$, $^4\mathrm{G}_{5/2}$, $^4\mathrm{G}_{5/2}$, $^4\mathrm{G}_{5/2}$, $^2\mathrm{D}_{5/2}$, $^2\mathrm{D}_{5/2}$, $^2\mathrm{D}_{5/2}$,

$^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{6}P_{7/2}$, $^{6}F_{7/2}$, $^{6}H_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{6}F_{9/2}$, $^{6}H_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{6}F_{11/2}$, $^{6}H_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{6}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}L_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{6}H_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}L_{15/2}$, $^{4}M_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}L_{17/2}$, $^{4}M_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{4}L_{19/2}$, $^{4}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}N_{19/2}$, $^{4}M_{21/2}$, $^{2}N_{21/2}$, $^{2}O_{21/2}$, $^{2}O_{23/2}$

$\underline{f}^{6}$ (295 LSJ levels)

$^{7}F_{0}$, $^{5}D_{0}$, $^{5}D_{0}$, $^{5}D_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{7}F_{1}$, $^{5}P_{1}$, $^{5}D_{1}$, $^{5}D_{1}$, $^{5}D_{1}$, $^{5}F_{1}$, $^{5}F_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{1}P_{1}$, $^{7}F_{2}$, $^{5}S_{2}$, $^{5}P_{2}$, $^{5}D_{2}$, $^{5}D_{2}$, $^{5}D_{2}$, $^{5}F_{2}$, $^{5}F_{2}$, $^{5}G_{2}$, $^{5}G_{2}$, $^{5}G_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{7}F_{3}$, $^{5}P_{3}$, $^{5}D_{3}$, $^{5}D_{3}$, $^{5}D_{3}$, $^{5}F_{3}$, $^{5}F_{3}$, $^{5}G_{3}$, $^{5}G_{3}$, $^{5}G_{3}$, $^{5}H_{3}$, $^{5}H_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{7}F_{4}$, $^{5}D_{4}$, $^{5}D_{4}$, $^{5}D_{4}$, $^{5}F_{4}$, $^{5}F_{4}$, $^{5}G_{4}$, $^{5}G_{4}$, $^{5}G_{4}$, $^{5}H_{4}$, $^{5}H_{4}$, $^{5}I_{4}$, $^{5}I_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{7}F_{5}$, $^{5}F_{5}$, $^{5}F_{5}$, $^{5}G_{5}$, $^{5}G_{5}$, $^{5}G_{5}$, $^{5}H_{5}$, $^{5}H_{5}$, $^{5}I_{5}$, $^{5}I_{5}$, $^{5}K_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{1}I_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{7}F_{6}$, $^{5}G_{6}$, $^{5}G_{6}$, $^{5}G_{6}$, $^{5}H_{6}$, $^{5}H_{6}$, $^{5}I_{6}$, $^{5}I_{6}$, $^{5}K_{6}$, $^{5}L_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{5}H_{7}$, $^{5}H_{7}$, $^{5}I_{7}$, $^{5}I_{7}$, $^{5}K_{7}$, $^{5}L_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}L_{7}$, $^{3}L_{7}$, $^{3}L_{7}$, $^{1}K_{7}$, $^{1}K_{7}$, $^{1}K_{7}$, $^{5}I_{8}$, $^{5}I_{8}$, $^{5}K_{8}$, $^{5}L_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}L_{8}$, $^{3}L_{8}$, $^{3}L_{8}$, $^{3}M_{8}$, $^{3}M_{8}$, $^{3}M_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{5}K_{9}$, $^{5}L_{9}$, $^{3}L_{9}$, $^{3}L_{9}$, $^{3}L_{9}$, $^{3}M_{9}$, $^{3}M_{9}$, $^{3}M_{9}$, $^{3}N_{9}$, $^{1}M_{9}$, $^{1}M_{9}$, $^{5}L_{10}$, $^{3}M_{10}$, $^{3}M_{10}$, $^{3}M_{10}$, $^{3}N_{10}$, $^{3}O_{10}$, $^{1}N_{10}$, $^{1}N_{10}$, $^{3}N_{11}$, $^{3}O_{11}$, $^{3}O_{12}$, $^{1}Q_{12}$

$\underline{f}^{7}$ (327 LSJ levels)

$^{6}D_{1/2}$, $^{6}F_{1/2}$, $^{4}P_{1/2}$, $^{4}P_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{2}S_{1/2}$, $^{2}S_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{6}P_{3/2}$, $^{6}D_{3/2}$, $^{6}F_{3/2}$, $^{6}G_{3/2}$, $^{4}S_{3/2}$, $^{4}S_{3/2}$, $^{4}P_{3/2}$, $^{4}P_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{6}P_{5/2}$, $^{6}D_{5/2}$, $^{6}F_{5/2}$, $^{6}G_{5/2}$, $^{6}H_{5/2}$, $^{4}P_{5/2}$, $^{4}P_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{8}S_{7/2}$, $^{6}P_{7/2}$, $^{6}D_{7/2}$, $^{6}F_{7/2}$, $^{6}G_{7/2}$, $^{6}H_{7/2}$, $^{6}I_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{6}D_{9/2}$, $^{6}F_{9/2}$, $^{6}G_{9/2}$, $^{6}H_{9/2}$, $^{6}I_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{6}F_{11/2}$, $^{6}G_{11/2}$, $^{6}H_{11/2}$, $^{6}I_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{6}G_{13/2}$, $^{6}H_{13/2}$, $^{6}I_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}L_{13/2}$, $^{4}L_{13/2}$, $^{4}L_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{6}H_{15/2}$, $^{6}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}L_{15/2}$, $^{4}L_{15/2}$, $^{4}L_{15/2}$, $^{4}M_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{6}I_{17/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}L_{17/2}$,

$^{4}L_{17/2}$, $^{4}L_{17/2}$, $^{4}M_{17/2}$, $^{4}N_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{4}L_{19/2}$, $^{4}L_{19/2}$, $^{4}L_{19/2}$, $^{4}M_{19/2}$, $^{4}N_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}N_{19/2}$, $^{2}N_{19/2}$, $^{4}M_{21/2}$, $^{4}N_{21/2}$, $^{2}N_{21/2}$, $^{2}N_{21/2}$, $^{2}O_{21/2}$, $^{4}N_{23/2}$, $^{2}O_{23/2}$, $^{2}Q_{23/2}$, $^{2}Q_{25/2}$

$\underline{f}^{8}$ (295 LSJ levels)

$^{7}F_{0}$, $^{5}D_{0}$, $^{5}D_{0}$, $^{5}D_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{7}F_{1}$, $^{5}P_{1}$, $^{5}D_{1}$, $^{5}D_{1}$, $^{5}D_{1}$, $^{5}F_{1}$, $^{5}F_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{1}P_{1}$, $^{7}F_{2}$, $^{5}S_{2}$, $^{5}P_{2}$, $^{5}D_{2}$, $^{5}D_{2}$, $^{5}D_{2}$, $^{5}F_{2}$, $^{5}F_{2}$, $^{5}G_{2}$, $^{5}G_{2}$, $^{5}G_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{7}F_{3}$, $^{5}P_{3}$, $^{5}D_{3}$, $^{5}D_{3}$, $^{5}D_{3}$, $^{5}F_{3}$, $^{5}F_{3}$, $^{5}G_{3}$, $^{5}G_{3}$, $^{5}G_{3}$, $^{5}H_{3}$, $^{5}H_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{1}F_{3}$, $^{7}F_{4}$, $^{5}D_{4}$, $^{5}D_{4}$, $^{5}D_{4}$, $^{5}F_{4}$, $^{5}F_{4}$, $^{5}G_{4}$, $^{5}G_{4}$, $^{5}G_{4}$, $^{5}H_{4}$, $^{5}H_{4}$, $^{5}I_{4}$, $^{5}I_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{7}F_{5}$, $^{5}F_{5}$, $^{5}F_{5}$, $^{5}G_{5}$, $^{5}G_{5}$, $^{5}G_{5}$, $^{5}H_{5}$, $^{5}H_{5}$, $^{5}I_{5}$, $^{5}I_{5}$, $^{5}K_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{7}F_{6}$, $^{5}G_{6}$, $^{5}G_{6}$, $^{5}G_{6}$, $^{5}H_{6}$, $^{5}H_{6}$, $^{5}I_{6}$, $^{5}I_{6}$, $^{5}K_{6}$, $^{5}L_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{5}H_{7}$, $^{5}H_{7}$, $^{5}I_{7}$, $^{5}I_{7}$, $^{5}K_{7}$, $^{5}L_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}L_{7}$, $^{3}L_{7}$, $^{3}L_{7}$, $^{1}K_{7}$, $^{1}K_{7}$, $^{1}K_{7}$, $^{5}I_{8}$, $^{5}I_{8}$, $^{5}K_{8}$, $^{5}L_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}L_{8}$, $^{3}L_{8}$, $^{3}L_{8}$, $^{3}M_{8}$, $^{3}M_{8}$, $^{3}M_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{5}K_{9}$, $^{5}L_{9}$, $^{3}L_{9}$, $^{3}L_{9}$, $^{3}L_{9}$, $^{3}M_{9}$, $^{3}M_{9}$, $^{3}M_{9}$, $^{3}N_{9}$, $^{1}M_{9}$, $^{1}M_{9}$, $^{5}L_{10}$, $^{3}M_{10}$, $^{3}M_{10}$, $^{3}M_{10}$, $^{3}N_{10}$, $^{3}O_{10}$, $^{1}N_{10}$, $^{1}N_{10}$, $^{3}N_{11}$, $^{3}O_{11}$, $^{3}O_{12}$, $^{1}Q_{12}$

$\underline{f}^{9}$ (198 LSJ levels)

$^{6}F_{1/2}$, $^{4}P_{1/2}$, $^{4}P_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{6}P_{3/2}$, $^{6}F_{3/2}$, $^{4}S_{3/2}$, $^{4}P_{3/2}$, $^{4}P_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{6}P_{5/2}$, $^{6}F_{5/2}$, $^{6}H_{5/2}$, $^{4}P_{5/2}$, $^{4}P_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{6}P_{7/2}$, $^{6}F_{7/2}$, $^{6}H_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{6}F_{9/2}$, $^{6}H_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{6}F_{11/2}$, $^{6}H_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{6}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}L_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{6}H_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}L_{15/2}$, $^{4}M_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}L_{17/2}$, $^{4}M_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{4}L_{19/2}$, $^{4}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}N_{19/2}$, $^{4}M_{21/2}$, $^{2}N_{21/2}$, $^{2}O_{21/2}$, $^{2}O_{23/2}$

$\underline{f}^{10}$ (107 LSJ levels)

$^{5}D_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{3}P_{0}$, $^{1}S_{0}$, $^{1}S_{0}$, $^{5}D_{1}$, $^{5}F_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}P_{1}$, $^{3}D_{1}$, $^{3}D_{1}$, $^{5}S_{2}$, $^{5}D_{2}$, $^{5}F_{2}$, $^{5}G_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}P_{2}$, $^{3}D_{2}$, $^{3}D_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{3}F_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{1}D_{2}$, $^{5}D_{3}$, $^{5}F_{3}$, $^{5}G_{3}$, $^{3}D_{3}$, $^{3}D_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}F_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{3}G_{3}$, $^{1}F_{3}$, $^{5}D_{4}$, $^{5}F_{4}$, $^{5}G_{4}$, $^{5}I_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}F_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}G_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{3}H_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{1}G_{4}$, $^{5}F_{5}$, $^{5}G_{5}$, $^{5}I_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}G_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}H_{5}$, $^{3}I_{5}$, $^{3}I_{5}$, $^{1}H_{5}$, $^{1}H_{5}$, $^{5}G_{6}$, $^{5}I_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}H_{6}$, $^{3}I_{6}$, $^{3}I_{6}$, $^{3}K_{6}$, $^{3}K_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{1}I_{6}$, $^{5}I_{7}$, $^{3}I_{7}$, $^{3}I_{7}$, $^{3}K_{7}$, $^{3}K_{7}$, $^{3}L_{7}$, $^{1}K_{7}$, $^{5}I_{8}$, $^{3}K_{8}$, $^{3}K_{8}$, $^{3}L_{8}$, $^{3}M_{8}$, $^{1}L_{8}$, $^{1}L_{8}$, $^{3}L_{9}$, $^{3}M_{9}$, $^{3}M_{10}$, $^{1}N_{10}$

$\underline{f}^{11}$ (41 LSJ levels)

$^{4}D_{1/2}$, $^{2}P_{1/2}$, $^{4}S_{3/2}$, $^{4}D_{3/2}$, $^{4}F_{3/2}$, $^{2}P_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{4}D_{5/2}$, $^{4}F_{5/2}$, $^{4}G_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$, $^{4}I_{9/2}$, $^{2}G_{9/2}$,

$^2G_{9/2}$, $^2H_{9/2}$, $^2H_{9/2}$, $^4G_{11/2}$, $^4I_{11/2}$, $^2H_{11/2}$, $^2H_{11/2}$, $^2I_{11/2}$, $^4I_{13/2}$, $^2I_{13/2}$, $^2K_{13/2}$, $^4I_{15/2}$, $^2K_{15/2}$, $^2L_{15/2}$, $^2L_{17/2}$

$\underline{f}^{12}$ (13 LSJ levels)
$^3P_0$, $^1S_0$, $^3P_1$, $^3P_2$, $^3F_2$, $^1D_2$, $^3F_3$, $^3F_4$, $^3H_4$, $^1G_4$, $^3H_5$, $^3H_6$, $^1I_6$

$\underline{f}^{13}$ (2 LSJ levels)
$^2F_{5/2}$, $^2F_{7/2}$

The level picture is a much more frugal description of the eigenstates. Not only are the number of basis elements that need to be considered much less than otherwise, but also the diagonalization is more efficient since it can be carried out within subspaces of shared J. One needs, however, to use adequate degeneracy factors in the relevant calculations.

In **qlanth** the function BasisLSJ can be used to retrieve the ordered basis that is used for the intermediate coupling description in terms of levels.

```
BasisLSJ::usage = "BasisLSJ[numE] returns the level basis LSJ. The function returns a list with each element representing the quantum numbers for each basis vector. Each element is of the form {SL (string in spectroscopic notation), J}.
The option ''AsAssociation'' can be set to True to return the basis as an association with the keys being the allowed J values. The default is False.
  ";
Options[BasisLSJ]={"AsAssociation"->False};
BasisLSJ[numE_,OptionsPattern[]] := Module[
  {Js, basis},
  (
    Js    = AllowedJ[numE];
    basis = BasisLSJMJ[numE,"AsAssociation"->False];
    basis = DeleteDuplicates[{#[[1]],#[[2]]}& /@ basis];
    If[OptionValue["AsAssociation"],
      (
        basis = Association @ Table[(J->Select[basis,#[[2]]==J&]),{J, Js}]
      )
    ];
    Return[basis];
  )
];
```

To obtain the blocks (indexed by J) representing the Hamiltonian in the level description, the function LevelSimplerEffectiveHamiltonian is provided in **qlanth**.

```
LevelSimplerEffectiveHamiltonian::usage = "
    LevelSimplerEffectiveHamiltonian[numE] is a variation of
    EffectiveHamiltonian that returns the diagonal JJ Hamiltonian
    blocks applying a simplifier and with simplifications adequate for
     the level description. The keys of the given association
    correspond to the different values of J that are possible for f^
    numE, the values are sparse array that are meant to be interpreted
     in the basis provided by BasisLSJ.
The option ''Simplifier'' is a list of symbols that are set to zero.
    At a minimum this has to include the crystal field parameters. By
    default this includes everything except the Slater parameters Fk
    and the spin orbit coupling ζ.
The option ''Export'' controls whether the resulting association is
    saved to disk, the default is True and the resulting file is saved
     to the ./hams/ folder. A hash is appended to the filename that
    corresponds to the simplifier used in the resulting expression. If
     the option ''Overwrite'' is set to False then these files may be
    used to quickly retrieve a previously computed case. The file is
    saved both in .m and .mx format.
The option ''PrependToFilename'' can be used to append a string to
    the filename to which the function may export to.
The option ''Return'' can be used to choose whether the function
    returns the matrix or not.
The option ''Overwrite'' can be used to overwrite the file if it
    already exists.";
```

```
Options[LevelSimplerEffectiveHamiltonian] = {
  "Export" -> True,
  "PrependToFilename" -> "",
  "Overwrite" -> False,
  "Return" -> True,
  "Simplifier" -> Join[
    {F0,\[Sigma]SS},
    cfSymbols,
    TSymbols,
    casimirSymbols,
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
  ]
};
LevelSimplerEffectiveHamiltonian[numE_Integer, OptionsPattern[]] :=
    Module[
  {thisHamAssoc, Js, fname,
  fnamemx, hash, simplifier},
  (
    simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
    hash       = Hash[simplifier];
    If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
    If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
    If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
    If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
    If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
    fname    = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".m"}];
    fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".mx"}];
    If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue["
    Overwrite"]],
    (
      If[OptionValue["Return"],
      (
        Which[FileExistsQ[fnamemx],
        (
          Echo["File " <> fnamemx <> " already exists, and option ''
    Overwrite'' is set to False, loading file ..."];
          thisHamAssoc=Import[fnamemx];
          Return[thisHamAssoc];
        ),
        FileExistsQ[fname],
        (
          Echo["File " <> fname <> " already exists, and option ''
    Overwrite'' is set to False, loading file ..."];
          thisHamAssoc=Import[fname];
          Echo["Exporting to file " <> fnamemx <> " for quicker
    loading."];
          Export[fnamemx,thisHamAssoc];
          Return[thisHamAssoc];
        )
        ]
      ),
      (
        Echo["File " <> fname <> " already exists, skipping ..."];
        Return[Null];
      )
      ]
    )
    ];
    Js           = AllowedJ[numE];
    thisHamAssoc = EffectiveHamiltonian[numE,
      "Set t2Switch"->True,
      "IncludeZeeman"->False,
      "ReturnInBlocks"->True
    ];
    thisHamAssoc = Diagonal[thisHamAssoc];
    thisHamAssoc = Map[SparseArray[ReplaceInSparseArray[#,simplifier
    ]]&,thisHamAssoc,{1}];
    thisHamAssoc = FreeHam[thisHamAssoc,numE];
    thisHamAssoc = AssociationThread[Js->thisHamAssoc];
    If[OptionValue["Export"],
```

```
      (
        Echo["Exporting to file " <> fname <> " and to " <> fnamemx];
        Export[fname,thisHamAssoc];
        Export[fnamemx,thisHamAssoc];
      )
    ];
    If[OptionValue["Return"],
      Return[thisHamAssoc],
      Return[Null]
    ];
  )
];
```

Whereas this description may be calculated without ever making explicit reference to M_J, in **qlanth** what is done is that the more verbose description associated with the $|LSJM_J\rangle$ basis is "downsized" to obtain the description in terms of levels. For this aim the following functions in **qlanth** are instrumental: `EigenLever`, `FreeHam`, `ListRepeater`, and `ListLever`.

The function `LevelSolver` can be used to calculate the level structure for given values of the parameters that one wishes to keep for the level description, which is often simply termed the *free-ion* part of the Hamiltonian.

```
LevelSolver::usage = "LevelSolver[numE, params] puts together (or retrieves from disk) the symbolic level Hamiltonian for the f^numE configuration and solves it for the given params returning the resultant energies and eigenstates.
If the option ''Return as states'' is set to False, then the function returns an association whose keys are values for J in f^numE, and whose values are lists with two elements. The first element being equal to the ordered basis for the corresponding subspace, given as a list of lists of the form {LS string, J}. The second element being another list of two elements, the first element being equal to the energies and the second being equal to the corresponding normalized eigenvectors. The energies given have been subtracted the energy of the ground state.
If the option ''Return as states'' is set to True, then the function returns a list with three elements. The first element is the global level basis for the f^numE configuration, given as a list of lists of the form {LS string, J}. The second element are the mayor LSJ components in the returned eigenstates. The third element is a list of lists with three elements, in each list the first element being equal to the energy, the second being equal to the value of J, and the third being equal to the corresponding normalized eigenvector (given as a row). The energies given have been subtracted the energy of the ground state, and the states have been sorted in order of increasing energy.
The following options are admitted:
  - ''Overwrite Hamiltonian'', if set to True the function will overwrite the symbolic Hamiltonian. Default is False.
  - ''Return as states'', see description above. Default is True.
  - ''Simplifier'', this is a list with symbols that are set to zero for defining the parameters kept in the level  description.
";
Options[LevelSolver] = {
  "Overwrite Hamiltonian" -> False,
  "Return as states" -> True,
  "Simplifier" -> Join[
    cfSymbols,
    TSymbols,
    casimirSymbols,
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
  ],
  "PrintFun" -> PrintTemporary
};
LevelSolver[numE_Integer, params0_Association, OptionsPattern[]] := Module[
  {ln, simplifier, simpleHam, basis,
  numHam, eigensys, startTime, endTime,
  diagonalTime, params=params0, globalBasis,
  eigenVectors, eigenEnergies, eigenJs,
  states, groundEnergy, allEnergies, PrintFun},
  (
    ln          = theLanthanides[[numE]];
```

```
    basis      = BasisLSJ[numE, "AsAssociation"->True];
    simplifier = OptionValue["Simplifier"];
    PrintFun   = OptionValue["PrintFun"];
    PrintFun["> LevelSolver for ",ln," with ",numE," f-electrons."];
    PrintFun["> Loading the symbolic level Hamiltonian ..."];
    simpleHam  = LevelSimplerEffectiveHamiltonian[numE,
      "Simplifier" -> simplifier,
      "Overwrite"  -> OptionValue["Overwrite Hamiltonian"]
    ];
    (* Everything that is not given is set to zero *)
    PrintFun["> Setting to zero every parameter not given ..."];
    params     = ParamPad[params, "PrintFun" -> PrintFun];
    PrintFun[params];
    (* Create the numeric hamiltonian *)
    PrintFun["> Replacing parameters in the J-blocks of the
    Hamiltonian to produce numeric arrays ..."];
    numHam     = N /@ Map[ReplaceInSparseArray[#, params]&, simpleHam
    ];
    Clear[simpleHam];
    (* Eigensolver *)
    PrintFun["> Diagonalizing the numerical Hamiltonian within each
    separate J-subspace ..."];
    startTime    = Now;
    eigensys     = Eigensystem /@ numHam;
    endTime      = Now;
    diagonalTime = QuantityMagnitude[endTime-startTime,"Seconds"];
    allEnergies  = Flatten[First/@Values[eigensys]];
    groundEnergy = Min[allEnergies];
    eigensys     = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys];
    eigensys     = Association@KeyValueMap[#1->{basis[#1],#2} &,
    eigensys];
    PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
    If[OptionValue["Return as states"],
      (
        PrintFun["> Padding the eigenvectors to correspond to the
    level basis ..."];
        eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values
    [#[[2, 2]] & /@ eigensys]];
        globalBasis   = Flatten[Values[basis], 1];
        eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
        eigenJs       = Flatten[KeyValueMap[ConstantArray[#1, Length
    [#2[[2, 2]]]] &, eigensys]];
        states        = Transpose[{eigenEnergies, eigenJs,
    eigenVectors}];
        states        = SortBy[states, First];
        eigenVectors  = Last /@ states;
        LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
    InputForm[#[[2]]]]) & /@ globalBasis;
        majorComponentIndices = Ordering[Abs[#]][[-1]] & /@
    eigenVectors;
        levelLabels           = LSJmultiplets[[majorComponentIndices
    ]];
        Return[{globalBasis, levelLabels, states}];
      ),
      Return[{basis, eigensys}]
    ];
  )
];
```

2.4 The coefficients of fractional parentage

In the 1920s and 1930s, when spectroscopic evidence was being studied to inform the emergent quantum mechanics, one conceptual tool that was put forward for the analysis of the complex spectra of ions [BG34] involved using the spectrum of an ion at one stage of ionization to understand another stage. For instance, using the fourth spectrum of oxygen (OIV) in order to understand the third spectrum (OIII) of the same element.

In 1943 Giulio Racah [Rac43] provided a useful extension to this idea. In addition of using the energies of one spectrum to span the energies of another, Racah extended this idea to the wavefunctions themselves, such that from configuration $\underline{\mathrm{f}}^{n-1}$ one can create the wavefunctions for $\underline{\mathrm{f}}^{n}$ with all the required antisymmetry and normalization conditions. In this approach, a given *daughter* term in $\underline{\mathrm{f}}^{n}$ has a number of *parent* terms in $\underline{\mathrm{f}}^{n-1}$, with the coefficients of fractional parentage determining how much of each parent is in the daughter

as a sum over parents

$$|\underline{\ell}^n \alpha LS\rangle = \sum_{\bar{\alpha}\bar{L}\bar{S}} \underbrace{\left(\underline{\ell}^{n-1}\bar{\alpha}\bar{L}\bar{S}\right\}\underline{\ell}^n \alpha LS\right)}_{\substack{\text{How much of parent } |\underline{\ell}^{n-1}\bar{\alpha}\bar{L}\bar{S}\rangle \\ \text{is in daughter} |\underline{\ell}^n \alpha LS\rangle}} \underbrace{|\left(\underline{\ell}^{n-1}\bar{\alpha}\bar{L}\bar{S}, \underline{\ell}\right) \alpha LS\rangle}_{\substack{\text{Couple an additional } \underline{\ell} \\ \text{to the parent } |\underline{\ell}^{n-1}\bar{\alpha}\bar{L}\bar{S}\rangle}} . \tag{14}$$

More importantly for **qlanth**, the coefficients of fractional parentage can be used to evaluate matrix elements of operators, such as in &qn-29, &qn-51, &qn-65, and &qn-41. These formulas realize a convenient calculation advantage: if one knows matrix elements in one configuration, then one can immediately calculate them in any other configuration.

In principle all the data that is needed in order to evaluate the matrix elements that **qlanth** uses can all be derived from coefficients of fractional parentage, tables of 6-j and 3-j coefficients, the LSUW labels for the terms in the $\underline{f}^n$ configurations, reduced matrix elements in $\underline{f}^3$ for the three-body operators, and reduced matrix elements in $\underline{f}^2$ for the magnetic interactions.

The data for the coefficients of fractional parentage we owe to [Vel00] from which the file `B1F_all.txt` originates, and which we use here to extract this useful "escalator" up the $\underline{f}^n$ configurations.

In **qlanth** the function `GenerateCFPTable` is used to parse the data contained in this file. From this data the association `CFP` is generated. This association has keys that represent LS terms for a configuration $\underline{f}^n$ and values that are lists which contain all the parent terms, together with the corresponding coefficients of fractional parentage.

```
GenerateCFPTable::usage = "GenerateCFPTable[] generates the table for the coefficients of fractional parentage. If the optional parameter ''Export'' is set to True then the resulting data is saved to ./data/CFPTable.m.
The data being parsed here is the file attachment B1F_ALL.TXT which comes from Velkov's thesis.";
Options[GenerateCFPTable] = {"Export" -> True};
GenerateCFPTable[OptionsPattern[]] := Module[
  {rawText, rawLines, leadChar, configIndex, line, daughter,
  lineParts, numberCode, parsedNumber, toAppend, CFPTablefname},
  (
    CleanWhitespace[string_]      := StringReplace[string, RegularExpression["\\s+"]->" "];
    AddSpaceBeforeMinus[string_] := StringReplace[string, RegularExpression["(?<!\\s)-"]->" -"];
    ToIntegerOrString[list_]      := Map[If[StringMatchQ[#, NumberString], ToExpression[#], #] &, list];
    CFPTable       = ConstantArray[{},7];
    CFPTable[[1]] = {{"2F",{"1S",1}}};

    (* Cleaning before processing is useful *)
    rawText  = Import[FileNameJoin[{moduleDir, "data", "B1F_ALL.TXT"}]];
    rawLines = StringTrim/@StringSplit[rawText,"\n"];
    rawLines = Select[rawLines,#!=""&];
    rawLines = CleanWhitespace/@rawLines;
    rawLines = AddSpaceBeforeMinus/@rawLines;

    Do[(
      (* the first character can be used to identify the start of a block *)
      leadChar=StringTake[line,{1}];
      (* ..FN, N is at position 50 in that line *)
      If[leadChar=="[",
      (
        configIndex=ToExpression[StringTake[line,{50}]];
        Continue[];
      )
      ];
      (* Identify which daughter term is being listed *)
      If[StringContainsQ[line,"[DAUGHTER TERM]"],
        daughter=StringSplit[line,"["][[1]];
        CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{daughter}];
        Continue[];
      ];
      (* Once we get here we are already parsing a row with coefficient data *)
      lineParts     = StringSplit[line," "];
```

```
      parent       = lineParts[[1]];
      numberCode   = ToIntegerOrString[lineParts[[3;;]]];
      parsedNumber = SquarePrimeToNormal[numberCode];
      toAppend     = {parent,parsedNumber};
      CFPTable[[configIndex]][[-1]] = Append[CFPTable[[configIndex
    ]][[-1]], toAppend]
    ),
    {line,rawLines}];
    If[OptionValue["Export"],
      (
      CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"
    }];
      Export[CFPTablefname, CFPTable];
      )
    ];
    Return[CFPTable];
  )
];
```

The coefficients of fractional parentage are traditionally only provided up to $\underline{\mathrm{f}}^7$ (such is the case in `B1f_all.txt`), tabulating these beyond $\underline{\mathrm{f}}^7$ would be redundant since the coefficients of fractional parentage beyond $\underline{\mathrm{f}}^7$ satisfy relationships with those below $\underline{\mathrm{f}}^7$. According to [NK63]

$$\left(\underline{\ell}^{(14-n)-1}\bar{\alpha}\bar{L}\bar{S}\right\}\underline{\ell}^{(14-n)}\alpha LS\Big) = \xi(-1)^{S+\bar{S}+L+\bar{L}-7/2}\sqrt{\frac{(n+1)[\![\bar{S}]\!][\![\bar{L}]\!]}{(14-n)[\![S]\!][\![L]\!]}}\left(\underline{\ell}^{n-1}\alpha LS\right\}\underline{\ell}^{n}\bar{\alpha}\bar{L}\bar{S}\Big)$$

$$\text{with } \xi = \begin{cases} 1 & \text{if } n \neq 6 \\ (-1)^{(\bar{\nu}-1)/2} & \text{if } n = 6 \end{cases}, \text{ and where } \bar{\boldsymbol{\nu}} \text{ is the seniority of } |\bar{\alpha}\bar{L}\bar{S}\rangle. \quad (15)$$

Under this relationship and phase convention, the matrix elements of operators pick up a global phase which depends on the rank of the operator, namely [NK63]:

$$\langle \underline{\mathrm{f}}^{14-n}\alpha SL\|\hat{U}^{(K)}\|\underline{\mathrm{f}}^{14-n}\alpha' S'L'\rangle = -(-1)^K\langle \underline{\mathrm{f}}^{n}\alpha SL\|\hat{U}^{(K)}\|\underline{\mathrm{f}}^{n}\alpha' S'L'\rangle \quad (16)$$

for a single tensor operator $\hat{U}^{(K)}$ of rank K, and

$$\langle \underline{\mathrm{f}}^{14-n}\alpha SL\|\hat{V}^{(1K)}\|\underline{\mathrm{f}}^{14-n}\alpha' S'L'\rangle = (-1)^K\langle \underline{\mathrm{f}}^{n}\alpha SL\|\hat{V}^{(1K)}\|\underline{\mathrm{f}}^{n}\alpha' S'L'\rangle \quad (17)$$

for a double tensor operator $\hat{V}^{(1K)}$ of rank 1 for spin and rank K for orbit.

2.5 Going beyond $\underline{\mathrm{f}}^7$

In most cases all matrix elements in **qlanth** are only calculated up to and including $\underline{\mathrm{f}}^7$. Beyond $\underline{\mathrm{f}}^7$ adequate changes of sign are enforced to take into account the equivalence that can be made between $\underline{\mathrm{f}}^n$ and $\underline{\mathrm{f}}^{14-n}$ as given by &qn-17 and &qn-16.

This is enforced when the function `EffectiveHamiltonian` is called. In there `HoleElectronConjugation` is the function responsible for enforcing a global sign flip for the following operators (or alternatively, to their accompanying coefficients):

$$\begin{aligned} &\zeta, B_q^{(k)} \\ &T^{(2)\prime}, T^{(3)}, T^{(4)}, T^{(6)}, T^{(7)}, T^{(8)} \\ &T^{(11)\prime}, T^{(12)}, T^{(13)}, T^{(14)}, T^{(15)}, T^{(16)}, T^{(17)}, T^{(18)}, T^{(19)}, \end{aligned} \quad (18)$$

$T^{(2)}$ and $T^{(11)}$ must be treated separately since they have a part that changes sign, and another that doesn't.

```
HoleElectronConjugation::usage = "HoleElectronConjugation[params]
    takes the parameters (as an association) that define a
    configuration and converts them so that they may be interpreted as
     corresponding to a complementary hole configuration. Some of this
     can be simply done by changing the sign of the model parameters.
    In the case of the effective three body interaction of T2 the
    relationship is more complex and is controlled by the value of the
     t2Switch variable.";
HoleElectronConjugation[params_] := Module[
  {newparams = params},
  (
    flipSignsOf = Join[{ζ} , cfSymbols, TSymbols];
    flipped = Table[
```

```
      (
        flipper -> - newparams[flipper]
      ),
      {flipper, flipSignsOf}
      ];
    nonflipped = Table[
      (
        flipper -> newparams[flipper]
      ),
      {flipper, Complement[Keys[newparams], flipSignsOf]}
      ];
    flippedParams = Association[Join[nonflipped, flipped]];
    flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
    Return[flippedParams];
  )
];
```

2.6 The J-J' block structure

Now that we know how the bases are ordered, we can already understand the structure of how the final Hamiltonian matrix representation in the $|LSJM_J\rangle$ basis is put together.

For a given configuration $\underline{f}^n$ and for each term $\hat{h}$ in the Hamiltonian, **qlanth** first calculates the matrix elements $\langle \alpha LSJM_J|\hat{h}|\alpha' L'S'J'M'_J\rangle$ so that for each interaction an association with keys of the form $\{J, J'\}$ is created. The values being rectangular arrays.

Fig-5 shows roughly this block structure for $\underline{f}^2$. In that figure, the shape of the rectangular blocks is determined by the fact that for $J = 0, 1, 2, 3, 4, 5, 6$ there are (2, 3, 15, 7, 27, 11, 26) corresponding basis states. As such, for example, the first row of blocks consists of blocks of size (2×2), (2×3), (2×15), (2×7), (2×27), (2×11), and (2×26).

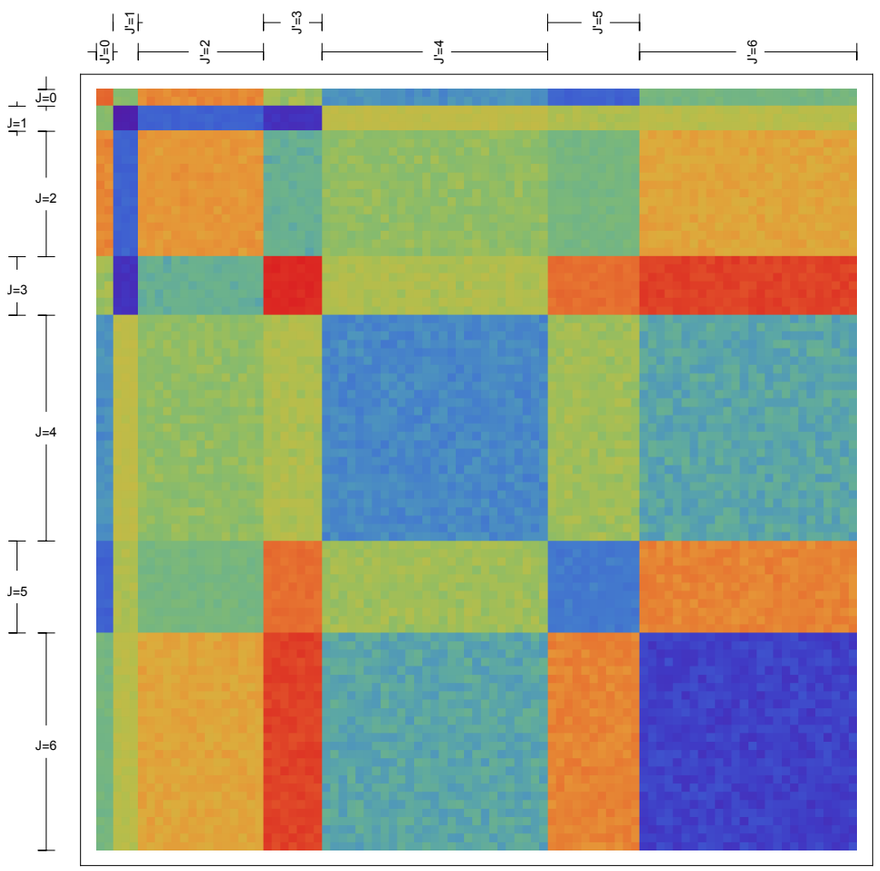


Figure 5: The J-J' block structure for $\underline{f}^2$

In **qlanth** these blocks are put together by the function JJBlockMatrix which adds together the contributions from the different terms in the Hamiltonian.

```
JJBlockMatrix::usage = "For given J, J' in the f^n configuration
    JJBlockMatrix[numE, J, J'] determines all the SL S'L' terms that
    may contribute to them and using those it provides the matrix
    elements <J, LS | H | J', LS'>. H having contributions from the
    following interactions: Coulomb, spin-orbit, spin-other-orbit,
```

```
    electrostatically-correlated-spin-orbit, spin-spin, three-body
    interactions, and crystal-field.";
Options[JJBlockMatrix] = {"Sparse"->True, "ChenDeltas"->False};
JJBlockMatrix[numE_, J_, Jp_, CFTable_, OptionsPattern[]] := Module[
  {NKSLJMs, NKSLJMps, NKSLJM, NKSLJMp,
  SLterm, SpLpterm,
  MJ, MJp,
  subKron, matValue, eMatrix},
  (
    NKSLJMs  = AllowedNKSLJMforJTerms[numE, J];
    NKSLJMps = AllowedNKSLJMforJTerms[numE, Jp];
    eMatrix =
      Table[
        (*Condition for a scalar matrix op*)
        SLterm   = NKSLJM[[1]];
        SpLpterm = NKSLJMp[[1]];
        MJ       = NKSLJM[[3]];
        MJp      = NKSLJMp[[3]];
        subKron  = (
            KroneckerDelta[J, Jp] *
            KroneckerDelta[MJ, MJp]
                );
        matValue =
          If[subKron==0,
            0,
              (
                ElectrostaticTable[{numE, SLterm, SpLpterm}] +
                ElectrostaticConfigInteraction[numE, {SLterm,
    SpLpterm}] +
                SpinOrbitTable[{numE, SLterm, SpLpterm, J}] +
                MagneticInteractions[{numE, SLterm, SpLpterm, J},
                  "ChenDeltas" -> OptionValue["ChenDeltas"]] +
                ThreeBodyTable[{numE, SLterm, SpLpterm}]
              )
          ];
        matValue += CFTable[{numE, SLterm, J, MJ, SpLpterm, Jp, MJp
    }];
        matValue,
      {NKSLJMp, NKSLJMps},
      {NKSLJM , NKSLJMs}
      ];
  If[OptionValue["Sparse"],
    eMatrix = SparseArray[eMatrix]
  ];
  Return[eMatrix]
)
];
```

Once these blocks have been calculated and saved to disk (in the folder `./hams/`) the function `EffectiveHamiltonian` takes them, assembles the arrays in block form, and finally flattens them to provide a sparse rank-2 array. These are the arrays that are finally diagonalized to find energies and eigenstates. Through options, this function can also return the Hamiltonian in block form, which is useful for the level description of the eigenstates.

```
EffectiveHamiltonian::usage = "EffectiveHamiltonian[numE] returns the
     Hamiltonian matrix for the f^numE configuration. The matrix is
    returned as a SparseArray.
The function admits an optional parameter ''FilenameAppendix'' which
    can be used to control which variant of the JJBlocks is used to
    assemble the matrix.
It also admits an optional parameter ''IncludeZeeman'' which can be
    used to include the Zeeman interaction. The default is False.
The option ''Set t2Switch'' can be used to toggle on or off setting
    the t2 selector automatically or not, the default is True, which
    replaces the parameter according to numE.
The option ''ReturnInBlocks'' can be used to return the matrix in
    block or flattened form. The default is to return it in flattened
    form.";
Options[EffectiveHamiltonian] = {
      "FilenameAppendix" -> "",
      "IncludeZeeman"    -> False,
      "Set t2Switch"     -> True,
      "ReturnInBlocks"   -> False,
      "OperatorBasis"    -> "Legacy"};
EffectiveHamiltonian[nf_, OptionsPattern[]] := Module[
```

```
  {numE, ii, jj, howManyJs, Js, blockHam, opBasis},
  (
    (*######################################*)
    ImportFun = ImportMZip;
    opBasis = OptionValue["OperatorBasis"];
    If[Not[MemberQ[{"Legacy", "MostlyOrthogonal", "Orthogonal"},
    opBasis]],
      Echo["Operator basis " <> opBasis <> " not recognized, using ''
    Legacy'' basis."];
      opBasis = "Legacy";
    ];
    If[opBasis == "Orthogonal",
      Echo["Operator basis ''Orthogonal'' not implemented yet,
    aborting ..."];
      Return[Null];
    ];
    (*######################################*)
    If[opBasis == "MostlyOrthogonal",
      (
        blockHam = EffectiveHamiltonian[nf,
        "OperatorBasis" -> "Legacy",
        "FilenameAppendix" -> OptionValue["FilenameAppendix"],
        "IncludeZeeman" -> OptionValue["IncludeZeeman"],
        "Set t2Switch" -> OptionValue["Set t2Switch"],
        "ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
        paramChanger = Which[
          nf < 7,
          <|
            F0 -> 1/91 (54 E1p+91 E0p+78 γp),
            F2 -> (15/392 *
              (
                140   E1p  +
                20020 E2p +
                1540  E3p  +
                770 αp -
                70 γp  +
                22 Sqrt[2] T2p -
                11 Sqrt[2] nf T2p t2Switch -
                11 Sqrt[2] (14-nf) T2p (1 - t2Switch)
              )
            ),
            F4 -> (99/490 *
              (
                70   E1p -
                9100 E2p +
                280  E3p +
                140 αp -
                35 γp  +
                4 Sqrt[2] T2p -
                2 Sqrt[2] nf T2p t2Switch -
                2 Sqrt[2] (14-nf) T2p (1-t2Switch)
              )
              ),
            F6 -> (5577/7000 *
              (
                20  E1p +
                700 E2p -
                140 E3p -
                70 αp -
                10 γp -
                2 Sqrt[2] T2p +
                Sqrt[2] nf T2p t2Switch +
                Sqrt[2] (14-nf) T2p (1-t2Switch)
              )
              ),
            ζ  -> ζ,
            α -> (5 αp)/4,
            β  -> -6  (5 αp + βp),
            γ -> 5/2 (2 βp + 5 γp),
            T2 -> 0
          |>,
        nf >= 7,
          <|
            F0 -> 1/91 (54 E1p+91 E0p+78 γp),
            F2 -> (15/392 *
              (
```

```
                140   E1p  +
                20020 E2p +
                1540  E3p  +
                770 αp -
                70 γp  +
                22 Sqrt[2] T2p -
                11 Sqrt[2] nf T2p
              )
            ),
            F4 -> (99/490 *
              (
                70   E1p -
                9100 E2p +
                280  E3p +
                140 αp -
                35 γp  +
                4 Sqrt[2] T2p -
                2 Sqrt[2] nf T2p
              )
              ),
            F6 -> (5577/7000 *
              (
                20  E1p +
                700 E2p -
                140 E3p -
                70 αp -
                10 γp -
                2 Sqrt[2] T2p +
                Sqrt[2] nf T2p
              )
              ),
            ζ  -> ζ,
            α -> (5 αp)/4,
            β  -> -6  (5 αp + βp),
            γ -> 5/2 (2 βp + 5 γp),
            T2 -> 0
          |>
        ];
        blockHamM0 = Which[
          OptionValue["ReturnInBlocks"] == False,
          ReplaceInSparseArray[blockHam, paramChanger],
          OptionValue["ReturnInBlocks"] == True,
          Map[ReplaceInSparseArray[#, paramChanger]&, blockHam, {2}]
        ];
        Return[blockHamM0];
      )
    ];
    (*######################################*)

    (*hole-particle equivalence enforcement*)
    numE = nf;
    allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2, T2p
    ,
      T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
      α, β, γ, B02, B04, B06, B12, B14, B16,
      B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16, S22
    ,
      S24, S26, S34, S36, S44, S46, S56, S66, T11p, T12, T14, T15,
    T16,
      T17, T18, T19, Bx, By, Bz};
    params0 = AssociationThread[allVars, allVars];
    If[nf > 7,
      (
        numE = 14 - nf;
        params = HoleElectronConjugation[params0];
        If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
      ),
      params = params0;
      If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
    ];
    (* Load symbolic expressions for LS,J,J' energy sub-matrices. *)
    emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
    OptionValue["FilenameAppendix"]];
    JJBlockMatrixTable = ImportFun[emFname];
    (*Patch together the entire matrix representation using J,J'
    blocks.*)
```

```
157     PrintTemporary["Patching JJ blocks ..."];
158     Js         = AllowedJ[numE];
159     howManyJs  = Length[Js];
160     blockHam   = ConstantArray[0, {howManyJs, howManyJs}];
161     Do[
162       blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[jj
        ]]}];,
163     {ii, 1, howManyJs},
164     {jj, 1, howManyJs}
165     ];
166     (* Once the block form is created flatten it *)
167     If[Not[OptionValue["ReturnInBlocks"]],
168       (blockHam = ArrayFlatten[blockHam];
169       blockHam  = ReplaceInSparseArray[blockHam, params];
170       ),
171       (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
        ,{2}];)
172     ];
173
174     If[OptionValue["IncludeZeeman"],
175       (
176         PrintTemporary["Including Zeeman terms ..."];
177         {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
        ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
178         blockHam += - teslaToKayser * (Bx * magx + By * magy + Bz *
        magz);
179       )
180     ];
181     Return[blockHam];
182   )
183 ];
```

In **qlanth** the reduced matrix elements of all operators, and the subsequent matrix elements of $\hat{\mathcal{H}}$ are calculated exactly. This is in contrast to what is done in older alternatives to **qlanth** such as linuxemp, in which calculations of reduced matrix elements were obtained from tables calculated with finite precision. To underscore this fact, Eqn-**??** shows an example of a J-J block as contained in **qlanth**.

2.7 Kramers' degeneracy

In the odd-electron cases, every energy is at least doubly degenerate. In **qlanth**, except in the case of the experimental data compiled for LaF_3, Kramers' degeneracy is given/expected explicitly.

3 Interactions

3.1 $\hat{\mathcal{H}}_{\mathrm{k}}$: kinetic energy

$$\hat{\mathcal{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 \text{ (kinetic energy of N v-electrons)} \tag{20}$$

Since our description is limited to a single configuration, the kinetic energy simply contributes a constant energy shift, and since all we care about are energy differences, then this term can be omitted from the analysis.

To interpret the range of energies that result from diagonalizing the semi-empirical Hamiltonian, it might be instructive, however, to note that this term imparts an energy of about $10\,\mathrm{eV} \approx 10^6\mathcal{K}$[16] to each electron.

3.2 $\hat{\mathcal{H}}_{\mathrm{e:sn}}$: the central field potential

$$\hat{\mathcal{H}}_{\mathrm{e:sn}} = -e^2\sum_{i=1}^{Z}\frac{1}{r_i} + e^2\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{Z-n}\frac{1}{r_{ij}}}_{\substack{\text{Repulsion between valence}\\ \text{and inner shell}\\ \text{electrons}}} \approx \sum_{i=1}^{n}V_{\mathrm{sn}}(r_i) \text{ (with Z = atomic No.)} \tag{21}$$

In principle, the sum over the Coulomb potential should extend over the nuclear charge and over all the electrons in the atom (not just the valence electrons). However, given the

[16] Note, (Kayser) $\mathcal{K} \equiv \mathrm{cm}^{-1}$, see section on units.

	$\vert^4\mathrm{F}_{1,-1}\rangle$	$\vert^4\mathrm{F}_{1,0}\rangle$	$\vert^4\mathrm{F}_{1,1}\rangle$
$\langle^4\mathrm{F}_{1,-1}\vert$	$2\alpha+\beta-\frac{B_0^{(2)}}{10}+\gamma$ $-\frac{\zeta}{2}+\frac{14F^{(0)}}{13}+\frac{43F^{(2)}}{195}+\frac{19F^{(4)}}{429}$ $-\frac{875F^{(6)}}{5577}+2m^{(0)}\sigma_{\mathrm{SS}}+\frac{61m^{(0)}}{12}$ $+4m^{(2)}\sigma_{\mathrm{SS}}+\frac{145m^{(2)}}{12}+\frac{50m^{(4)}\sigma_{\mathrm{SS}}}{11}$ $+\frac{1805m^{(4)}}{132}+\frac{43P^{(2)}}{1080}+\frac{19P^{(4)}}{2376}$ $-\frac{875P^{(6)}}{30888}$	$-\frac{\sqrt{3}B_1^{(2)}}{10}-\frac{1}{10}i\sqrt{3}S_2^{(2)}$	$-\frac{1}{5}\sqrt{\frac{3}{2}}B_2^{(2)}-\frac{1}{5}i\sqrt{\frac{3}{2}}S_2^{(2)}$
$\langle^4\mathrm{F}_{1,0}\vert$	$-\frac{\sqrt{3}B_1^{(2)}}{10}+\frac{1}{10}i\sqrt{3}S_2^{(2)}$	$2\alpha+\beta+\frac{B_0^{(2)}}{5}+\gamma$ $-\frac{\zeta}{2}+\frac{14F^{(0)}}{13}+\frac{43F^{(2)}}{195}+\frac{19F^{(4)}}{429}$ $-\frac{875F^{(6)}}{5577}+2m^{(0)}\sigma_{\mathrm{SS}}+\frac{61m^{(0)}}{12}$ $+4m^{(2)}\sigma_{\mathrm{SS}}+\frac{145m^{(2)}}{12}+\frac{50m^{(4)}\sigma_{\mathrm{SS}}}{11}$ $+\frac{1805m^{(4)}}{132}+\frac{43P^{(2)}}{1080}+\frac{19P^{(4)}}{2376}$ $-\frac{875P^{(6)}}{30888}$	$\frac{\sqrt{3}B_1^{(2)}}{10}+\frac{1}{10}i\sqrt{3}S_2^{(2)}$
$\langle^4\mathrm{F}_{1,1}\vert$	$-\frac{1}{5}\sqrt{\frac{3}{2}}B_2^{(2)}+\frac{1}{5}i\sqrt{\frac{3}{2}}S_2^{(2)}$	$\frac{\sqrt{3}B_1^{(2)}}{10}-\frac{1}{10}i\sqrt{3}S_2^{(2)}$	$2\alpha+\beta-\frac{B_0^{(2)}}{10}+\gamma$ $-\frac{\zeta}{2}+\frac{14F^{(0)}}{13}+\frac{43F^{(2)}}{195}+\frac{19F^{(4)}}{429}$ $-\frac{875F^{(6)}}{5577}+2m^{(0)}\sigma_{\mathrm{SS}}+\frac{61m^{(0)}}{12}$ $+4m^{(2)}\sigma_{\mathrm{SS}}+\frac{145m^{(2)}}{12}+\frac{50m^{(4)}\sigma_{\mathrm{SS}}}{11}$ $+\frac{1805m^{(4)}}{132}+\frac{43P^{(2)}}{1080}+\frac{19P^{(4)}}{2376}$ $-\frac{875P^{(6)}}{30888}$

(19)

shell structure of the atom, the lanthanide ions "see" the nuclear charge as shielded by a xenon core. Since every closed shell is a singlet, having spherical symmetry, these shields are like spherical shells surrounding the nucleus.

The precise form of $V_{\text{sn}}(r_i)$ is not of our concern here; all that matters is that we assume that it is spherically symmetric so that we can justify the separation of radial and angular parts of the wavefunctions.

3.3 $\hat{\mathcal{H}}_{\text{e:e}}$: e:e repulsion

$$\hat{\mathcal{H}}_{\text{e:e}} = \sum_{i>j}^{n,n} \frac{e^2}{\|\vec{r}_i - \vec{r}_j\|} = \sum_{k=0,2,4,6} F^{(k)} \hat{f}_k = \sum_{k=0,1,2,3} E_k \hat{e}_k \tag{22}$$

This term is the first we will not discard. Calculating this term for the $\underline{\text{f}}^n$ configurations was one of the contributions from Slater, as such the parameters we use to write it up are called *Slater integrals*. After the analysis from Slater, Giulio Racah contributed further to the analysis of this term [Rac49]. The insight that Racah had was that if in a given operator one identifies the parts in it that transform accordingly to the different symmetry groups present in the problem, then calculating the necessary matrix element in all $\underline{\text{f}}^n$ configurations can be greatly simplified.

The functions used in `qlanth` to compute these LS-reduced matrix elements [17] are `Electrostatic` and `fsubk`. In addition to these, the LS-reduced matrix elements of the tensor operators $\hat{C}^{(k)}$ and $\hat{U}^{(k)}$ are also needed. These functions are based in equations 12.16 and 12.17 from [Cow81] as specialized for the case of electrons belonging to a single $\underline{\text{f}}^n$ configuration. By default this term is computed in terms of $F^{(k)}$ Slater integrals, but it can also be computed in terms of the E_k Racah parameters, the functions `EtoF` and `FtoE` are useful for going from one representation to the other.

$$\langle \underline{\text{f}}^n \alpha^{2S+1}\text{L} \| \hat{\mathcal{H}}_{\text{e:e}} \| \underline{\text{f}}^n \alpha'^{2S'+1}\text{L'} \rangle = \sum_{k=0,2,4,6} {F^{(f)}}_k(n, \alpha LS, \alpha' L' S') \tag{23}$$

where

$$f_k(n, \alpha LS, \alpha' L' S') = \frac{1}{2}\delta(S,S')\delta(L,L')\langle \underline{\text{f}} \| \hat{C}^{(k)} \| \underline{\text{f}} \rangle^2 \times$$
$$\left\{ \frac{1}{2L+1} \sum_{\alpha'' L''} \langle \underline{\text{f}}^n \alpha'' L'' S \| \hat{U}^{(k)} \| \underline{\text{f}}^n \alpha LS \rangle \langle \underline{\text{f}}^n \alpha'' L'' S \| \hat{U}^{(k)} \| \underline{\text{f}}^n \alpha' LS \rangle - \delta(\alpha, \alpha') \frac{n\,(4\underline{\text{f}} + 2 - n)}{(2\underline{\text{f}}+1)(4\underline{\text{f}}+1)} \right\}. \tag{24}$$

```
Electrostatic::usage = "Electrostatic[{numE, NKSL, NKSLp}] returns
    the LS reduced matrix element for repulsion matrix element for
    equivalent electrons. See equation 2-79 in Wybourne (1965). The
    option ''Coefficients'' can be set to ''Slater'' or ''Racah''. If
    set to ''Racah'' then E_k parameters and e^k operators are assumed
    , otherwise the Slater integrals F^k and operators f_k. The
    default is ''Slater''.";
Options[Electrostatic] = {"Coefficients" -> "Slater"};
Electrostatic[{numE_, NKSL_, NKSLp_}, OptionsPattern[]] := Module[
  {fsub0, fsub2, fsub4, fsub6,
   esub0, esub1, esub2, esub3,
   fsup0, fsup2, fsup4, fsup6,
   eMatrixVal, orbital},
  (
    orbital = 3;
    Which[
      OptionValue["Coefficients"] == "Slater",
      (
        fsub0 = fsubk[numE, orbital, NKSL, NKSLp, 0];
        fsub2 = fsubk[numE, orbital, NKSL, NKSLp, 2];
        fsub4 = fsubk[numE, orbital, NKSL, NKSLp, 4];
        fsub6 = fsubk[numE, orbital, NKSL, NKSLp, 6];
        eMatrixVal = fsub0*F0 + fsub2*F2 + fsub4*F4 + fsub6*F6;
      ),
      OptionValue["Coefficients"] == "Racah",
      (
        fsup0 = fsupk[numE, orbital, NKSL, NKSLp, 0];
        fsup2 = fsupk[numE, orbital, NKSL, NKSLp, 2];
        fsup4 = fsupk[numE, orbital, NKSL, NKSLp, 4];
        fsup6 = fsupk[numE, orbital, NKSL, NKSLp, 6];
```

[17] An LS-reduced matrix element is ...

```
        esub0 = fsup0;
        esub1 = 9/7*fsup0 +   1/42*fsup2 +   1/77*fsup4 +  1/462*
    fsup6;
        esub2 =              143/42*fsup2 - 130/77*fsup4 + 35/462*
    fsup6;
        esub3 =               11/42*fsup2 + 4/77*fsup4   -  7/462*
    fsup6;
        eMatrixVal = esub0*E0 + esub1*E1 + esub2*E2 + esub3*E3;
      )
    ];
    Return[eMatrixVal];
  )
];
```

```
fsubk::usage = "fsubk[numE, orbital, SL, SLp, k] gives the Slater
    integral f_k for the given configuration and pair of SL terms. See
     equation 12.17 in TASS.";
fsubk[numE_, orbital_, NKSL_, NKSLp_, k_] := Module[
  {terms, S, L, Sp, Lp,
  termsWithSameSpin, SL,
  fsubkVal, spinMultiplicity,
  prefactor, summand1, summand2},
  (
    {S, L}   = FindSL[NKSL];
    {Sp, Lp} = FindSL[NKSLp];
    terms    = AllowedNKSLTerms[numE];
    (* sum for summand1 is over terms with same spin *)
    spinMultiplicity  = 2*S + 1;
    termsWithSameSpin = StringCases[terms, ToString[spinMultiplicity]
     ~~ __];
    termsWithSameSpin = Flatten[termsWithSameSpin];
    If[Not[{S, L} == {Sp, Lp}],
      Return[0]
    ];
    prefactor = 1/2 * Abs[Ck[orbital, k]]^2;
    summand1 = Sum[(
        ReducedUkTable[{numE, orbital, SL, NKSL,  k}] *
        ReducedUkTable[{numE, orbital, SL, NKSLp, k}]
        ),
      {SL, termsWithSameSpin}
    ];
    summand1 = 1 / TP0[L] * summand1;
    summand2 = (
      KroneckerDelta[NKSL, NKSLp] *
        (numE *(4*orbital + 2 - numE)) /
        ((2*orbital + 1) * (4*orbital + 1))
      );
    fsubkVal = prefactor*(summand1 - summand2);
    Return[fsubkVal];
  )
];
```

```
EtoF::usage = "EtoF[E0, E1, E2, E3] calculates the Slater integral
    parameters {F0, F2, F4, F6} corresponding to the given Racah
    parameters {E0, E1, E2, E3}. This is the inverse of the FtoE
    function.";
EtoF[E0_, E1_, E2_, E3_] := Module[
  {F0, F2, F4, F6},
  (
    F0 = 1/7      (7 E0 + 9 E1);
    F2 = 75/14    (E1 + 143 E2 + 11 E3);
    F4 = 99/7     (E1 - 130 E2 + 4 E3);
    F6 = 5577/350 (E1 + 35 E2 - 7 E3);
    Return[{F0, F2, F4, F6}];
  )
];
```

```
FtoE::usage = "FtoE[F0, F2, F4, F6] calculates the Racah parameters {
    E0, E1, E2, E3} corresponding to the given Slater integrals.
See eqn. 2-80 in Wybourne.
Note that in that equation the subscripted Slater integrals are used
    but since this function assumes the the input values are
    superscripted Slater integrals, it is necessary to convert them
    using Dk.";
FtoE[F0_, F2_, F4_, F6_] := Module[
```

```
  {E0, E1, E2, E3},
  (
    E0 = (F0 - 10*F2/Dk[2] - 33*F4/Dk[4] - 286*F6/Dk[6]);
    E1 = (70*F2/Dk[2] + 231*F4/Dk[4] + 2002*F6/Dk[6])/9;
    E2 = (F2/Dk[2] - 3*F4/Dk[4] + 7*F6/Dk[6])/9;
    E3 = (5*F2/Dk[2] + 6*F4/Dk[4] - 91*F6/Dk[6])/3;
    Return[{E0, E1, E2, E3}];
  )
];
```

3.4 $\hat{\mathcal{H}}_{\text{s:o}}$: spin-orbit

The spin-orbit interaction arises from the interaction of the magnetic moment of the electron and the magnetic field that its orbital motion generates. In terms of the central potential $V_{\text{s:n}}$, the spin-orbit term for a single electron is

$$\hat{h}_{\text{s:o}} = \frac{\hbar^2}{2m_e^2c^2}\left(\frac{1}{r}\frac{\mathrm{d}V_{\text{s:n}}}{\mathrm{d}r}\right)\hat{l}\cdot\hat{s} \coloneqq \zeta(r)\hat{l}\cdot\hat{s}. \tag{25}$$

Adding this term for all the n valence electrons, and replacing $\zeta(r)$ by its radial average ζ then gives

$$\hat{\mathcal{H}}_{\text{s:o}} = \zeta\sum_i^n \hat{l}_i\cdot\hat{s}_i. \tag{26}$$

From equations 2-106 to 2-109 in Wybourne [Wyb63] the matrix elements we need are given by

$$\begin{aligned}\langle\alpha LSJM_J|\hat{\mathcal{H}}_{\text{s:o}}|\alpha' L'S'J'M_{J'}\rangle &= \zeta\delta(J,J')\delta(M_J,M_{J'})\langle\alpha LSJM_J|\sum_i^n \hat{l}_i\cdot\hat{s}_i|\alpha' L'S'JM_J\rangle\\ &= \zeta\delta(J,J')\delta(M_J,M_{J'})(-1)^{J+L+S'}\begin{Bmatrix}L & L' & 1\\ S' & S & J\end{Bmatrix}\langle\alpha LS|\sum_i^n \hat{l}_i\cdot\hat{s}_i|\alpha' L'S'\rangle\\ = \zeta\delta(J,J')\delta(M_J,M_{J'})(-1)^{J+L+S'}&\begin{Bmatrix}L & L' & 1\\ S' & S & J\end{Bmatrix}\sqrt{\underline{\ell}(\underline{\ell}+1)(2\underline{\ell}+1)}\langle\alpha LS\|\hat{V}^{(11)}\|\alpha' L'S'\rangle,\end{aligned} \tag{27}$$

where $\hat{V}^{(11)}$ is a double tensor operator of rank one over spin and orbital parts defined as

$$\hat{V}^{(11)} = \sum_{i=1}^n\left(\hat{s}\hat{u}^{(1)}\right)_i, \tag{28}$$

in which the rank on the spin operator $\hat{s}$ has been omitted, and the rank of the orbital tensor operator given explicitly as 1.

In **qlanth** the reduced matrix elements for this double tensor operator are calculated by `ReducedV1k` and stored in a static association called `ReducedV1kTable`. The reduced matrix elements of this operator are calculated using equation 2-101 from Wybourne [Wyb65]:

$$\begin{aligned}\langle\underline{\ell}^n\psi\|\hat{V}^{(1k)}\|\underline{\ell}^n\psi'\rangle = \langle\underline{\ell}^n\alpha LS\|\hat{V}^{(1k)}\|\underline{\ell}^n\alpha' L'S'\rangle = n\sqrt{\underline{s}(\underline{s}+1)(2\underline{s}+1)}\sqrt{[\![S]\!][\![L]\!][\![S']\!][\![L']\!]}\times\\ \sum_{\bar{\psi}}(-1)^{\bar{S}+\bar{L}+S+L+\underline{\ell}+\underline{s}+k+1}\left(\psi\left\{\left|\bar{\psi}\right.\right)\left(\left.\bar{\psi}\right\}\right|\psi'\right)\begin{Bmatrix}S & S' & 1\\ \underline{s} & \underline{s} & \bar{S}\end{Bmatrix}\begin{Bmatrix}L & L' & k\\ \underline{\ell} & \underline{\ell} & \bar{L}\end{Bmatrix}\end{aligned} \tag{29}$$

In this expression the sum over $\bar{\psi}$ depends on (ψ,ψ') and is over all the states in $\underline{\ell}^{n-1}$ which are common parents to both ψ and ψ'. Also note that in the equation above, since our concern are $\underline{\text{f}}$-electron configurations, we have $\underline{\ell}=3$ and $\underline{s}=\frac{1}{2}$.

```
ReducedV1k::usage = "ReducedV1k[n, SL, SpLp, k] gives the reduced
    matrix element of the spherical tensor operator V^(1k). See
    equation 2-101 in Wybourne 1965.";
ReducedV1k[numE_, SL_, SpLp_, k_] := Module[
  {Vk1, S, L, Sp, Lp,
  Sb, Lb, spin, orbital,
  cfpSL, cfpSpLp,
  SLparents, SpLpparents,
  commonParents, prefactor},
  (
    {spin, orbital} = {1/2, 3};
```

```
    {S, L}         = FindSL[SL];
    {Sp, Lp}       = FindSL[SpLp];
    cfpSL          = CFP[{numE, SL}];
    cfpSpLp        = CFP[{numE, SpLp}];
    SLparents      = First /@ Rest[cfpSL];
    SpLpparents    = First /@ Rest[cfpSpLp];
    commonParents = Intersection[SLparents, SpLpparents];
    Vk1 = Sum[(
        {Sb, Lb} = FindSL[\[Psi]b];
        Phaser[(Sb + Lb + S + L + orbital + k - spin)] *
        CFPAssoc[{numE, SL, \[Psi]b}] *
        CFPAssoc[{numE, SpLp, \[Psi]b}] *
        SixJay[{S, Sp, 1}, {spin, spin, Sb}] *
        SixJay[{L, Lp, k}, {orbital, orbital, Lb}]
      ),
    {\[Psi]b, commonParents}
    ];
    prefactor = numE * Sqrt[spin * (spin + 1) * TPO[spin, S, L, Sp, Lp] ];
    Return[prefactor * Vk1];
  )
];
```

These reduced matrix elements are then used by the function `SpinOrbit`.

```
SpinOrbit::usage = "SpinOrbit[numE, SL, SpLp, J] returns the LSJ reduced matrix element ζ <SL, J|L.S|SpLp, J>. These are given as a function of ζ. This function requires that the association ReducedV1kTable be defined.
See equations 2-106 and 2-109 in Wybourne (1965). Equivalently see eqn. 12.43 in TASS.";
SpinOrbit[numE_, SL_, SpLp_, J_] := Module[
  {S, L, Sp, Lp, orbital, sign, prefactor, val},
  (
    orbital    = 3;
    {S, L}     = FindSL[SL];
    {Sp, Lp}   = FindSL[SpLp];
    prefactor = Sqrt[orbital * (orbital+1) * (2*orbital+1)] *
                SixJay[{L, Lp, 1}, {Sp, S, J}];
    sign       = Phaser[J + L + Sp];
    val        = sign * prefactor * ζ * ReducedV1kTable[{numE, SL, SpLp, 1}];
    Return[val];
  )
];
```

3.5 $\hat{\mathcal{H}}_{\mathcal{SO}(3)}, \hat{\mathcal{H}}_{\mathcal{G}_2}, \hat{\mathcal{H}}_{\mathcal{SO}(7)}$: electrostatic configuration interaction

These are the first terms where we take into account the contributions from *configuration-interaction*. Rajnak and Wybourne [RW63] showed that *configuration-interaction* of the electrostatic interactions corresponding to two-electron excitations from $\underline{f}^n$ can be represented through the Casimir operators of the groups $\mathcal{SO}(3)$, $\mathcal{G}_2$, and $\mathcal{SO}(7)$. This borrowed from an earlier insight of Trees [Tre52], who realized that an addition of a term proportional to $L(L+1)$ improved the energy calculations for the second spectrum of manganese (Mn-II) and the third spectrum of iron (Fe-III).

One of these Casimir operators is the familiar $\hat{L}^2$ from $\mathcal{SO}(3)$. In analogy to $\hat{L}^2$ in which the quantum number L can be used to determine the eigenvalues, in the cases of $\hat{\mathcal{H}}_{\mathcal{G}_2}$ the necessary state label is the U label of the LS term, and in the case of $\hat{\mathcal{H}}_{\mathcal{SO}(7)}$ the necessary label is W. If $\Lambda_{\mathcal{G}_2}(U)$ is used to note the eigenvalue of the Casimir operator of $\mathcal{G}_2$ corresponding to label U, and $\Lambda_{\mathcal{SO}(7)}(W)$ the eigenvalue corresponding to state label W, then the matrix elements of $\hat{\mathcal{H}}_{\mathcal{SO}(3)}$, $\hat{\mathcal{H}}_{\mathcal{G}_2}$ and $\hat{\mathcal{H}}_{\mathcal{SO}(7)}$ are diagonal in all quantum numbers (see Rajnak and Wybourne [RW63]) and are given by

$$\langle \underline{\ell}^n \alpha S L J M_J | \hat{\mathcal{H}}_{\mathcal{SO}(3)} | \underline{\ell}^n \alpha' S' L' J' M'_J \rangle = \alpha \delta(\alpha S L J M_J, \alpha' S' L' J' M'_J) L(L+1) \tag{30}$$

$$\langle \underline{\ell}^n U \alpha S L J M_J | \hat{\mathcal{H}}_{\mathcal{G}_2} | \underline{\ell}^n U \alpha' S' L' J' M'_J \rangle = \beta \delta(\alpha S L J M_J, \alpha' S' L' J' M'_J) \Lambda_{\mathcal{G}_2}(U) \tag{31}$$

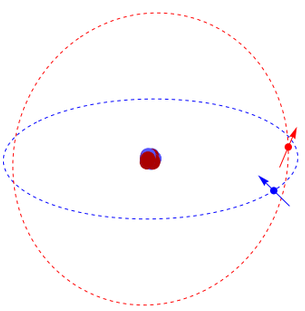
$$\langle \underline{\ell}^n W \alpha S L J M_J | \hat{\mathcal{H}}_{\mathcal{SO}(7)} | \underline{\ell}^n W \alpha' S' L' J' M_J \rangle = \gamma \delta(\alpha S L J M_J, \alpha' S' L' J' M'_J) \Lambda_{\mathcal{SO}(7)}(W) \tag{32}$$

In **qlanth** the role of $\Lambda_{\mathcal{SO}(7)}(W)$ is played by the function `GSO7W`, the role of $\Lambda_{\mathcal{G}_2}(U)$ by `GG2U`, and the role of $\Lambda_{\mathcal{SO}(3)}(L)$ by `CasimirSO3`. These are used by `CasimirG2`, `CasimirSO3`, and `CasimirSO7` which find the corresponding U, W, L labels to the LS terms provided to them. Finally, the function `ElectrostaticConfigInteraction` puts them together.

```
ElectrostaticConfigInteraction::usage = "
    ElectrostaticConfigInteraction[numE, {SL, SpLp}] returns the
    matrix element for configuration interaction as approximated by
    the Casimir operators of the groups R3, G2, and R7. SL and SpLp
    are strings that represent terms under LS coupling.";
ElectrostaticConfigInteraction[numE_, {SL_, SpLp_}] := Module[
  {S, L, val},
  (
    If[
      Or[numE == 1, numE==13],
      Return[0];
    ];
    {S, L} = FindSL[SL];
    val = (
      If[SL == SpLp,
        CasimirSO3[{SL, SL}] +
        CasimirSO7[{SL, SL}] +
        CasimirG2[{SL, SL}],
        0
      ]
      );
    ElectrostaticConfigInteraction[numE, {S, L}] = val;
    Return[val];
  )
];
```

3.6 $\hat{\mathcal{H}}_{\text{s:s-s:oo}}$: *spin-spin* and *spin-other-orbit*

The calculation of the $\hat{\mathcal{H}}_{\text{s:s-s:oo}}$ is qualitatively different from the previous ones. The previous ones were self-contained in the sense that the reduced matrix elements that we require we also computed on our own. In the case of the interactions that follow from here, we use values from literature for reduced matrix elements either in $\underline{f}^2$ or in $\underline{f}^3$ and then we "pull" them up for all $\underline{f}^n$ configurations with help of the coefficients of fractional parentage.



The analysis of *spin-other-orbit*, and the *spin-spin* contributions used in **qlanth** is that of Judd, Crosswhite, and Crosswhite [JCC68]. Much as the spin-orbit effect can be extracted from the Dirac equation as a relativistic correction, the multi-electron spin-orbit effects can be derived from the Breit operator $\hat{\mathcal{H}}_B$ [BS57] which is a term added to the relativistic description of a many-particle system in order to account for retardation of the electromagnetic field

$$\hat{\mathcal{H}}_B = -\frac{1}{2}e^2 \sum_{i>j} \left[(\alpha_i \cdot \alpha_j) \frac{1}{r_{ij}} + (\alpha_i \cdot \vec{r}_{ij})(\alpha_j \cdot \vec{r}_{ij}) \frac{1}{r_{ij}^3} \right]. \tag{33}$$

When this operator is expanded in powers of v/c, a number of non-relativistic inter-electron interactions result. Two of them are the *spin-other-orbit* and *spin-spin* interactions. As usual, the radial part of the Hamiltonian is averaged, which in this case gives appearance to the Marvin integrals

$$m^{(k)} := \frac{e^2\hbar^2}{8m^2c^2} \langle (nl)^2 | \frac{r_<^k}{r_>^{k+3}} | (nl)^2 \rangle. \tag{34}$$

With these, the expression for the *spin-spin* term within the single configuration description is [JCC68]

$$\hat{\mathcal{H}}_{s:s} = -2 \sum_{i \neq j} \sum_k m^{(k)} \sqrt{(k+1)(k+2)(2k+3)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle \langle \underline{\ell} \| C^{(k+2)} \| \underline{\ell} \rangle \left\{ \hat{w}_i^{(1,k)} \hat{w}_j^{(1,k+2)} \right\}^{(2,2)0} \tag{35}$$

and the one for *spin-other-orbit*

$$\hat{\mathcal{H}}_{s:oo} = \sum_{i\neq j}\sum_{k}\sqrt{(k+1)(2\underline{\ell}+k+2)(2\underline{\ell}-k)}\times \left[\left\{\hat{w}_i^{(0,k+1)}\hat{w}_j^{(1,k)}\right\}^{(11)0}\left\{m^{(k-1)}\langle\underline{\ell}\|C^{(k+1)}\|\underline{\ell}\rangle^2 + 2m^{(k)}\langle\underline{\ell}\|C^{(k)}\|\underline{\ell}\rangle^2\right\} + \left\{\hat{w}_i^{(0,k)}\hat{w}_j^{(1,k+1)}\right\}^{(11)0}\left\{m^{(k)}\langle\underline{\ell}\|C^{(k)}\|\underline{\ell}\rangle^2 + 2m^{(k-1)}\langle\underline{\ell}\|C^{(k+1)}\|\underline{\ell}\rangle^2\right\}\right]. \quad (36)$$

In the expressions above $\hat{w}_i^{(\kappa,k)}$ is a double tensor operator of rank κ over spin, of rank k over orbit, and acting on electron i. It is defined by its reduced matrix elements as

$$\langle\underline{\ell}\|\hat{w}^{(\kappa,k)}\|\underline{\ell}\rangle = \sqrt{[\![\kappa]\!][\![k]\!]}. \quad (37)$$

The explicit complexity of the above expressions can be somewhat reduced by identifying them with the scalar part of two new double tensors $\hat{\mathcal{T}}_0^{(11)}$ and $\hat{\mathcal{T}}_0^{(22)}$ such that

$$\sqrt{5}\hat{\mathcal{T}}_0^{(22)} \coloneqq \hat{\mathcal{H}}_{\text{s:s}} \quad (38)$$

$$-\sqrt{3}\hat{\mathcal{T}}_0^{(11)} \coloneqq \hat{\mathcal{H}}_{\text{s:oo}}. \quad (39)$$

In terms of which the reduced matrix elements in the $|LSJ\rangle$ basis can be obtained by

$$\langle\gamma SLJ|\hat{\mathcal{H}}|\gamma' S'L'J'\rangle = \delta(J,J')\begin{Bmatrix} S' & L' & J \\ L & S & t\end{Bmatrix}\langle\gamma SL\|\hat{\mathcal{T}}^{(tt)}\|\gamma' S'L'\rangle. \quad (40)$$

This above relationship is the one effectively used in **qlanth** in the functions SpinSpin and SOOandECSO.

```
SpinSpin::usage = "SpinSpin[n, SL, SpLp, J] returns the matrix
    element <|SL,J|spin-spin|SpLp,J|> for the spin-spin operator
    within the configuration f^n. This matrix element is independent
    of MJ. This is obtained by querying the relevant reduced matrix
    element from the association T22Table, putting in the adequate
    phase, and 6-j symbol.
This is calculated according to equation (3) in ''Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic
    Interactions for f Electrons.'' Physical Review 169, no. 1 (1968):
     130.''
''.
";
SpinSpin[numE_, SL_, SpLp_, J_] := Module[
  {S, L, Sp, Lp, α, val},
  (
    α = 2;
    {S, L} = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            T22Table[{numE, SL, SpLp}]
          );
    Return[val]
  )
];
```

```
SOOandECSO::usage = "SOOandECSO[n, SL, SpLp, J] returns the matrix
    element <|SL,J|spin-spin|SpLp,J|> for the combined effects of the
    spin-other-orbit interaction and the electrostatically-correlated-
    spin-orbit (which originates from configuration interaction
    effects) within the configuration f^n. This matrix element is
    independent of MJ. This is obtained by querying the relevant
    reduced matrix element by querying the association
    SOOandECSOLSTable and putting in the adequate phase and 6-j symbol
    . The SOOandECSOLSTable puts together the reduced matrix elements
    from three operators.
This is calculated according to equation (3) in ''Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic
    Interactions for f Electrons.'' Physical Review 169, no. 1 (1968):
     130.''.
";
SOOandECSO[numE_, SL_, SpLp_, J_] := Module[
```

```
  {S, Sp, L, Lp, α, val},
  (
    α = 1;
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            SOOandECSOLSTable[{numE, SL, SpLp}]
          );
    Return[val];
  )
];
```

For two-electron operators such as these, the matrix elements in $\underline{\mathrm{f}}^n$ are related to those in $\underline{\mathrm{f}}^{n-1}$ by equation 4 in Judd *et al.* [JCC68]

$$\langle \underline{\mathrm{f}}^n \psi \| \hat{\mathcal{T}}^{(tt)} \| \underline{\mathrm{f}}^n \psi' \rangle = \frac{n}{n-2} \sum_{\bar{\psi},\bar{\psi}'} (-1)^{\bar{S}+\bar{L}+\underline{\imath}+\underline{\ell}+S'+L'} \sqrt{[\![S]\!][\![S']\!][\![L]\!][\![L']\!]} \times \left(\psi\{|\bar{\psi}\right)\left(\psi'\{|\bar{\psi}'\right) \begin{Bmatrix} S & t & S' \\ \bar{S}' & \underline{\imath} & \bar{S} \end{Bmatrix} \begin{Bmatrix} L & t & L' \\ \bar{L}' & \underline{\ell} & \bar{L} \end{Bmatrix} \langle \underline{\mathrm{f}}^{n-1} \bar{\psi} \| \hat{\mathcal{T}}^{(tt)} \| \underline{\mathrm{f}}^{n-1}] \bar{\psi}' \rangle, \quad (41)$$

where the sum runs over the terms $\bar{\psi}$ and $\bar{\psi}'$ in $\underline{\mathrm{f}}^{n-1}$ which are parents common to ψ and ψ'. Using these the matrix elements of $\hat{\mathcal{T}}^{(11)}$ and $\hat{\mathcal{T}}^{(22)}$ in $\underline{\mathrm{f}}^2$ can be used to compute all the reduced matrix elements in $\underline{\mathrm{f}}^n$. These could then be used together with Eqn-40 to obtain the matrix elements of $\hat{\mathcal{H}}_{\mathrm{s:s}}$ and $\hat{\mathcal{H}}_{\mathrm{s:oo}}$. This is done for $\hat{\mathcal{H}}_{\mathrm{s:s}}$, but not for $\hat{\mathcal{H}}_{\mathrm{s:oo}}$, because this term is traditionally computed (with a slight modification) at the same time as the electrostically-correlated-spin-orbit (see next section).

These equations are implemented in **qlanth** through the following functions: `GenerateT22Table`, `ReducedT22infn`, `ReducedT22inf2`, `ReducedT11inf2`. Where `ReducedT22inf2` and `ReducedT11inf2` provide the reduced matrix elements for $\hat{\mathcal{T}}^{(11)}$ and $\hat{\mathcal{T}}^{(22)}$ in $\underline{\mathrm{f}}^2$ as provided in table II of [JCC68].

```
GenerateT22Table::usage = "GenerateT22Table[nmax] generates the LS
    reduced matrix elements for the double tensor operator T22 in f^n
    up to n=nmax. If the option ''Export'' is set to true then the
    resulting association is saved to the data folder. The values for
    n=1 and n=2 are taken from ''Judd, BR, HM Crosswhite, and Hannah
    Crosswhite. ''Intra-Atomic Magnetic Interactions for f Electrons
    .'' Physical Review 169, no. 1 (1968): 130.'', and the values for
    n>2 are calculated recursively using equation (4) of that same
    paper.
This is an intermediate step to the calculation of the reduced matrix
     elements of the spin-spin operator.";
Options[GenerateT22Table] = {"Export" -> True, "Progress" -> True};
GenerateT22Table[nmax_Integer, OptionsPattern[]] := (
  If[And[OptionValue["Progress"], frontEndAvailable],
    (
      numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
    numE]]^2, {numE, 1, nmax}]];
      counters = Association[Table[numE->0, {numE, 1, nmax}]];
      totalIters = Total[Values[numItersai[[1;;nmax]]]];
      template1 = StringTemplate["Iteration `numiter` of `totaliter`"
    ];
      template2 = StringTemplate["`remtime` min remaining"];template3
     = StringTemplate["Iteration speed = `speed` ms/it"];
      template4 = StringTemplate["Time elapsed = `runtime` min"];
      progBar = PrintTemporary[
        Dynamic[
          Pane[
            Grid[{{Superscript["f", numE]},
                  {template1[<|"numiter"->numiter, "totaliter"->
    totalIters|>]},
                  {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                  {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
    ]], 0.1]|>]},
                  {template3[<|"speed"->Round[QuantityMagnitude[Now-
    startTime, "ms"]/(numiter), 0.01]|>]},
                  {ProgressIndicator[Dynamic[numiter], {1, totalIters
    }]}},
```

```
                    Frame->All],
                    Full,
                    Alignment->Center]
                ]
              ];
    )
  ];
  T22Table = <||>;
  startTime = Now;
  numiter = 1;
  Do[
    (
      numiter+= 1;
      T22Table[{numE, SL, SpLp}] = Which[
        numE==1,
        0,
        numE==2,
        SimplifyFun[ReducedT22inf2[SL, SpLp]],
        True,
        SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
      ];
    ),
  {numE, 1, nmax},
  {SL,   AllowedNKSLTerms[numE]},
  {SpLp, AllowedNKSLTerms[numE]}
  ];
  If[And[OptionValue["Progress"],frontEndAvailable],
    NotebookDelete[progBar]
  ];
  If[OptionValue["Export"],
    (
      fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"}];
      Export[fname, T22Table];
    )
  ];
  Return[T22Table];
);
```

```
ReducedT22infn::usage = "ReducedT22infn[n, SL, SpLp] calculates the
    reduced matrix element of the T22 operator for the f^n
    configuration corresponding to the terms SL and SpLp.
This is done by using equation (4) of ''Judd, BR, HM Crosswhite, and
    Hannah Crosswhite. ''Intra-Atomic Magnetic Interactions for f
    Electrons.'' Physical Review 169, no. 1 (1968): 130.''
";
ReducedT22infn[numE_, SL_, SpLp_] := Module[
  {spin, orbital, t, idx1, idx2, S, L,
  Sp, Lp, cfpSL, cfpSpLp, parentSL,
  parentSpLp, Sb, Lb, Tnkk, phase, Sbp, Lbp},
  (
    {spin, orbital} = {1/2, 3};
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    t = 2;
    cfpSL    = CFP[{numE, SL}];
    cfpSpLp  = CFP[{numE, SpLp}];
    Tnkk = Sum[(
      parentSL   = cfpSL[[idx2, 1]];
      parentSpLp = cfpSpLp[[idx1, 1]];
      {Sb, Lb}   = FindSL[parentSL];
      {Sbp, Lbp} = FindSL[parentSpLp];
      phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
      (
        phase *
        cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
        SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
        SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
        T22Table[{numE - 1, parentSL, parentSpLp}]
      )
    ),
    {idx1, 2, Length[cfpSpLp]},
    {idx2, 2, Length[cfpSL]}
    ];
    Tnkk *= numE / (numE - 2) * Sqrt[TPO[S,Sp,L,Lp]];
    Return[Tnkk];
  )
```

```
];
```

```
ReducedT22inf2::usage = "ReducedT22inf2[SL, SpLp] returns the reduced
    matrix element of the scalar component of the double tensor T22
    for the terms SL, SpLp in f^2.
Data used here for m0, m2, m4 is from Table I of Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. Intra-Atomic Magnetic
    Interactions for f Electrons. Physical Review 169, no. 1 (1968):
    130.
";
ReducedT22inf2[SL_, SpLp_] := Module[
  {statePosition, PsiPsipStates, m0, m2, m4, Tkk2m},
  (
    T22inf2 = <|
    {"3P", "3P"} -> -12 M0 - 24 M2 - 300/11 M4,
    {"3P", "3F"} -> 8/Sqrt[3] (3 M0 + M2 - 100/11 M4),
    {"3F", "3F"} -> 4/3 Sqrt[14] (-M0 + 8 M2 - 200/11 M4),
    {"3F", "3H"} -> 8/3 Sqrt[11/2] (2 M0 - 23/11 M2 - 325/121 M4),
    {"3H", "3H"} -> 4/3 Sqrt[143] (M0 - 34/11 M2 - (1325/1573) M4)
    |>;
    Which[
      MemberQ[Keys[T22inf2],{SL,SpLp}],
        Return[T22inf2[{SL,SpLp}]],
      MemberQ[Keys[T22inf2],{SpLp,SL}],
        Return[T22inf2[{SpLp,SL}]],
      True,
        Return[0]
    ]
  )
];
```

```
Reducedt11inf2::usage = "Reducedt11inf2[SL, SpLp] returns the reduced
    matrix element in f^2 of the double tensor operator t11 for the
    corresponding given terms {SL, SpLp}.
Values given here are those from Table VII of ''Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic
    Interactions for f Electrons.'' Physical Review 169, no. 1 (1968):
    130.''
";
Reducedt11inf2[SL_, SpLp_] := Module[
  {t11inf2},
  (
    t11inf2 = <|
      {"1S", "3P"} -> -2 P0 - 105 P2 - 231 P4 - 429 P6,
      {"3P", "3P"} -> -P0 - 45 P2 - 33 P4 + 1287 P6,
      {"3P", "1D"} -> Sqrt[15/2] (P0 + 32 P2 - 33 P4 - 286 P6),
      {"1D", "3F"} -> Sqrt[10] (-P0 - 9/2 P2 + 66 P4 - 429/2 P6),
      {"3F", "3F"} -> Sqrt[14] (-P0 + 10 P2 + 33 P4 + 286 P6),
      {"3F", "1G"} -> Sqrt[11] (P0 - 20 P2 + 32 P4 - 104 P6),
      {"1G", "3H"} -> Sqrt[10] (-P0 + 55/2 P2 - 23 P4 - 65/2 P6),
      {"3H", "3H"} -> Sqrt[55] (-P0 + 25 P2 + 51 P4 + 13 P6),
      {"3H", "1I"} -> Sqrt[13/2] (P0 - 21 P4 - 6 P6)
      |>;
    Which[
      MemberQ[Keys[t11inf2],{SL,SpLp}],
        Return[t11inf2[{SL,SpLp}]],
      MemberQ[Keys[t11inf2],{SpLp,SL}],
        Return[t11inf2[{SpLp,SL}]],
      True,
        Return[0]
    ]
  )
];
```

3.7 $\hat{\mathcal{H}}_{\text{ecs:o}}$: *electrostatically-correlated-spin-orbit*

In the same paper [JCC68] that describes the *spin-spin* and *spin-other-orbit* interactions, consideration is also given to the emergence of additional corrections due to configuration interaction as described by the following operator (which is what results from the application of perturbation theory to *second* order) (page. 134 of [JCC68])

$$\hat{\mathcal{H}}_{\text{ci}} = -\sum_{\chi}\sum_{i}\frac{1}{E_\chi}\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right)|\chi\rangle\langle\chi|\hat{\mathfrak{C}} - \frac{1}{E_\chi}\hat{\mathfrak{C}}|\chi\rangle\langle\chi|\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \tag{42}$$

where $\xi(r_h)(\hat{\underline{s}}_h \cdot \hat{\underline{\ell}}_h)$ is the customary spin-orbit interaction, E_χ is the energy of state $|\chi\rangle$, i is a label for the valence electrons, $\hat{\mathfrak{C}}$ stands for the Coulomb interaction, and $|\chi\rangle$ are states in the configurations with which one is "interacting". Since this term includes both the electrostatic term and the spin-orbit one, this is called the *electrostatically-correlated-spin-orbit* interaction.

This operator can be identified with the scalar component of a double tensor operator of rank 1 both for the spin and orbital parts of the wavefunction

$$\hat{\mathcal{H}}_{\text{ci}} = -\sqrt{3}\,\hat{t}_0^{(11)}. \tag{43}$$

Judd *et al.* [JCC68] then go on to list the reduced matrix elements of this operator in the $\underline{\text{f}}^2$ configuration. When this is done the Marvin integrals $m^{(k)}$ appear again, but a second set of parameters, the *pseudo-magnetic* parameters $P^{(k)}$, is also necessary

$$P^{(k)} = 6\sum_{f'} \frac{\zeta_{ff'}}{E_{ff'}} R^{(k)}(ff, ff') \text{ for } k = 0, 2, 4, 6. \tag{44}$$

Where f stands for an $\underline{\text{f}}$-electron radial eigenfunction, and f' similarly but for a configuration different from $\underline{\text{f}}^n$. And where

$$\zeta_{ff'} \coloneqq \langle f|\xi(r)|f'\rangle \tag{45}$$

$$R^{(k)}(ff, ff') \coloneqq e^2\langle f_1 f_2|\frac{r_<^k}{r_>^{k+1}}|f_1 f_2'\rangle. \tag{46}$$

In the semi-empirical approach embodied by **qlanth**, calculating these quantities *ab-initio* is not the objective, they are instead to be defined from experiments. Nonetheless, not only these expressions give theoretical justification to the model, but they also serve to justify the ratios between different orders of these quantities, their relative importance, or their sign. Consequently, both the set of three $m^{(k)}$ and the set of $P^{(k)}$ ultimately rely on a single free parameter each. Such parsimony is desirable given the large number of parameters (about 20) that the Hamiltonian ends up having.

Judd *et al.* further note that $P^{(0)}$ is proportional to the spin orbit operator, and as such its effect is absorbed by the standard spin-orbit parameter ζ. They also developed an alternative approach based on group theory arguments. They put together the *spin-other-orbit* and the *electrostatically-correlated-spin-orbit* as a sum of operators $\hat{z}_i$ with useful transformation rules

$$\langle\psi\|\hat{\mathcal{T}}^{(11)} + \hat{t}^{(11)}\|\psi'\rangle = \sum a_i\langle\psi\|\hat{z}_i\|\psi'\rangle. \tag{47}$$

At this stage a subtle point needs to be raised. As Judd points out, in the sum above, the term $\hat{z}_{13}$ that contributes with a tensorial character equal to that of the regular spin-orbit operator. As such, if the goal is obtaining a parametric Hamiltonian that can be fit with uncorrelated parameters, it is then necessary to subtract this part from $\hat{\mathcal{T}}^{(11)} + \hat{t}^{(11)}$. This point was clarified by Chen *et al.* [Che+08]. Because of this, the final form of the operator contributing both to *spin-other-orbit* and the *electrostatically-correlated-spin-orbit* is

$$\hat{\mathcal{H}}_{\text{s:oo}} + \hat{\mathcal{H}}_{\text{ecs:o}} = \hat{\mathcal{T}}^{(11)} + \hat{t}^{(11)} - \frac{1}{6}a_{13}\hat{z}_{13} \tag{48}$$

where

$$a_{13} = -33m^{(0)} + 3m^{(2)} + \frac{15}{11}m^{(4)} - 6P^{(0)} + \frac{3}{2}\left(\frac{35}{225}P^{(2)} + \frac{77}{1089}P^{(4)} + \frac{25}{1287}P^{(6)}\right). \tag{49}$$

In **qlanth** the contributions from *spin-spin*, *spin-other-orbit*, and *electrostatically-correlated-spin-orbit* are put together by the function `MagneticInteractions`. That function queries precomputed values from two associations `SpinSpinTable` and `SOOandECSOTable`. In turn these two associations are generated by the functions `GenerateSpinOrbitTable` and `GenerateSOOandECSOTable`. Note that both *spin-spin* and *spin-other-orbit* end up contributing through $m^{(k)}$, however there doesn't seem to be consensus about adding them together, as such **qlanth** allows including or excluding the *spin-spin* contribution, this is done with a control parameter σ_{SS} (1 for including, 0 for excluding).

```
MagneticInteractions::usage = "MagneticInteractions[{numE, SL, SLp, J
    }] returns the matrix element of the magnetic interaction between
    the terms SL and SLp in the f^numE configuration for the given
    value of J. The interaction is given by the sum of the spin-spin,
    the spin-other-orbit, and the electrostatically-correlated-spin-
    orbit interactions.
```

```
The part corresponding to the spin-spin interaction is provided by
    SpinSpin[{numE, SL, SLp, J}].
The part corresponding to SOO and ECSO is provided by the function
    SOOandECSO[{numE, SL, SLp, J}].
The option ''ChenDeltas'' can be used to include or exclude the Chen
    deltas from the calculation. The default is to exclude them. If
    this option is used, then the chenDeltas association needs to be
    loaded into the session with LoadChenDeltas[].";
Options[MagneticInteractions] = {"ChenDeltas" -> False};
MagneticInteractions[{numE_, SL_, SLp_, J_}, OptionsPattern[]] :=
    Module[
  {key, ss, sooandecso, total,
  S, L, Sp, Lp, phase, sixjay,
  M0v, M2v, M4v,
  P2v, P4v, P6v},
  (
    key        = {numE, SL, SLp, J};
    ss         = \[Sigma]SS * SpinSpinTable[key];
    sooandecso = SOOandECSOTable[key];
    total = ss + sooandecso;
    total = SimplifyFun[total];
    If[
      Not[OptionValue["ChenDeltas"]],
      Return[total]
    ];
    (* In the type A errors the wrong values are different *)
    If[MemberQ[Keys[chenDeltas["A"]], {numE, SL, SLp}],
      (
        {S, L}   = FindSL[SL];
        {Sp, Lp} = FindSL[SLp];
        phase    = Phaser[Sp + L + J];
        sixjay   = SixJay[{Sp, Lp, J}, {L, S, 1}];
        {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SL,
    SLp}]["wrong"];
        total    = (
                     phase * sixjay *
                     (
                       M0v*M0 + M2v*M2 + M4v*M4 +
                       P2v*P2 + P4v*P4 + P6v*P6
                     )
                   );
        total    = wChErrA * total + (1 - wChErrA) * (ss + sooandecso
    )
      )
    ];
    (* In the type B errors the wrong values are zeros all around *)
    If[MemberQ[chenDeltas["B"], {numE, SL, SLp}],
      (
        total  = (1 - wChErrB) * (ss + sooandecso)
      )
    ];
    Return[total];
  )
];
```

```
GenerateSpinOrbitTable::usage = "GenerateSpinOrbitTable[nmax]
    computes the matrix elements for the spin-orbit interaction for f^
    n configurations up to n = nmax. The function returns an
    association whose keys are lists of the form {n, SL, SpLp, J}. If
    ''Export'' is set to True, then the result is exported to the data
     folder. It requires ReducedV1kTable to be defined.";
Options[GenerateSpinOrbitTable] = {"Export" -> True};
GenerateSpinOrbitTable[nmax_Integer:7, OptionsPattern[]] := Module[
  {numE, J, SL, SpLp, exportFname},
  (
    SpinOrbitTable =
      Table[
        {numE, SL, SpLp, J} -> SpinOrbit[numE, SL, SpLp, J],
      {numE, 1, nmax},
      {J, MinJ[numE], MaxJ[numE]},
      {SL,   Map[First, AllowedNKSLforJTerms[numE, J]]},
      {SpLp, Map[First, AllowedNKSLforJTerms[numE, J]]}
      ];
    SpinOrbitTable = Association[SpinOrbitTable];

    exportFname = FileNameJoin[{moduleDir, "data", "SpinOrbitTable.m"
```

```
    }];
    If[OptionValue["Export"],
      (
        Echo["Exporting to file " <> ToString[exportFname]];
        Export[exportFname, SpinOrbitTable];
      )
    ];
    Return[SpinOrbitTable];
  )
];
```

```
GenerateSOOandECSOTable::usage = "GenerateSOOandECSOTable[nmax]
    generates the reduced matrix elements in the |LSJ> basis for the (
    spin-other-orbit + electrostatically-correlated-spin-orbit)
    operator. It returns an association where the keys are of the form
     {n, SL, SpLp, J}. If the option ''Export'' is set to True then
    the resulting object is saved to the data folder. Since this is a
    scalar operator, there is no MJ dependence. This dependence only
    comes into play when the crystal field contribution is taken into
    account.";
Options[GenerateSOOandECSOTable] = {"Export"->False};
GenerateSOOandECSOTable[nmax_, OptionsPattern[]] := (
  SOOandECSOTable = <||>;
  Do[
    SOOandECSOTable[{numE, SL, SpLp, J}] = (SOOandECSO[numE, SL, SpLp
    , J] /. Prescaling);,
  {numE, 1, nmax},
  {J, MinJ[numE], MaxJ[numE]},
  {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
  {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
  ];
  If[OptionValue["Export"],
  (
    fname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
    Export[fname, SOOandECSOTable];
  )
  ];
  Return[SOOandECSOTable];
);
```

The function `GenerateSpinSpinTable` calls the function `SpinSpin` over all possible combinations of the arguments $\{n, SL, S'L', J\}$. In turn the function `SpinSpin` queries the precomputed values of the double tensor $\hat{\mathcal{T}}^{(22)}$ which are stored in the association `T22Table`.

```
GenerateSpinSpinTable::usage = "GenerateSpinSpinTable[nmax] generates
     the reduced matrix elements in the |LSJ> basis for the spin-spin
    operator. It returns an association where the keys are of the form
     {numE, SL, SpLp, J}. If the option ''Export'' is set to True then
     the resulting object is saved to the data folder. Since this is a
     scalar operator, there is no MJ dependence. This dependence only
    comes into play when the crystal field contribution is taken into
    account.";
Options[GenerateSpinSpinTable] = {"Export"->False};
GenerateSpinSpinTable[nmax_, OptionsPattern[]] :=
  (
    SpinSpinTable = <||>;
    PrintTemporary[Dynamic[numE]];
    Do[
      SpinSpinTable[{numE, SL, SpLp, J}] = (SpinSpin[numE, SL, SpLp,
    J]);,
    {numE, 1, nmax},
    {J, MinJ[numE], MaxJ[numE]},
    {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
    {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
    ];
    If[OptionValue["Export"],
    (fname = FileNameJoin[{moduleDir, "data", "SpinSpinTable.m"}];
      Export[fname, SpinSpinTable];
      )
    ];
    Return[SpinSpinTable];
  );
```

```
SpinSpin::usage = "SpinSpin[n, SL, SpLp, J] returns the matrix element <|SL,J|spin-spin|SpLp,J|> for the spin-spin operator within the configuration f^n. This matrix element is independent of MJ. This is obtained by querying the relevant reduced matrix element from the association T22Table, putting in the adequate phase, and 6-j symbol.
This is calculated according to equation (3) in ''Judd, BR, HM Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic Interactions for f Electrons.'' Physical Review 169, no. 1 (1968): 130.''
''.
";
SpinSpin[numE_, SL_, SpLp_, J_] := Module[
  {S, L, Sp, Lp, α, val},
  (
    α = 2;
    {S, L} = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            T22Table[{numE, SL, SpLp}]
          );
    Return[val]
  )
];
```

The association `T22Table` is computed by the function `GenerateT22Table`. This function populates `T22Table` with keys of the form $\{n, SL, S'L'\}$. It does this by using the function `ReducedT22inf2` in the base case of $\underline{f}^2$, and `ReducedT22infn` for configurations above $\underline{f}^2$. When `ReducedT22infn` is called, the sum in &qn-41 is carried out using $t = 2$. When `ReducedT22inf2` is called, the reduced matrix elements from [JCC68] are used.

```
GenerateT22Table::usage = "GenerateT22Table[nmax] generates the LS reduced matrix elements for the double tensor operator T22 in f^n up to n=nmax. If the option ''Export'' is set to true then the resulting association is saved to the data folder. The values for n=1 and n=2 are taken from ''Judd, BR, HM Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic Interactions for f Electrons.'' Physical Review 169, no. 1 (1968): 130.'', and the values for n>2 are calculated recursively using equation (4) of that same paper.
This is an intermediate step to the calculation of the reduced matrix elements of the spin-spin operator.";
Options[GenerateT22Table] = {"Export" -> True, "Progress" -> True};
GenerateT22Table[nmax_Integer, OptionsPattern[]] := (
  If[And[OptionValue["Progress"], frontEndAvailable],
    (
      numItersai = Association[Table[numE->Length[AllowedNKSLTerms[numE]]^2, {numE, 1, nmax}]];
      counters = Association[Table[numE->0, {numE, 1, nmax}]];
      totalIters = Total[Values[numItersai[[1;;nmax]]]];
      template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
      template2 = StringTemplate["`remtime` min remaining"];template3 = StringTemplate["Iteration speed = `speed` ms/it"];
      template4 = StringTemplate["Time elapsed = `runtime` min"];
      progBar = PrintTemporary[
        Dynamic[
          Pane[
            Grid[{{Superscript["f", numE]},
                  {template1[<|"numiter"->numiter, "totaliter"->totalIters|>]},
                  {template4[<|"runtime"->Round[QuantityMagnitude[UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                  {template2[<|"remtime"->Round[QuantityMagnitude[UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"]], 0.1]|>]},
                  {template3[<|"speed"->Round[QuantityMagnitude[Now-startTime, "ms"]/(numiter), 0.01]|>]},
                  {ProgressIndicator[Dynamic[numiter], {1, totalIters}]}},
                  Frame->All],
                  Full,
                  Alignment->Center]
                ]
```

```
                ];
    )
  ];
  T22Table = <||>;
  startTime = Now;
  numiter = 1;
  Do[
    (
      numiter+= 1;
      T22Table[{numE, SL, SpLp}] = Which[
        numE==1,
        0,
        numE==2,
        SimplifyFun[ReducedT22inf2[SL, SpLp]],
        True,
        SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
      ];
    ),
  {numE, 1, nmax},
  {SL,   AllowedNKSLTerms[numE]},
  {SpLp, AllowedNKSLTerms[numE]}
  ];
  If[And[OptionValue["Progress"],frontEndAvailable],
    NotebookDelete[progBar]
  ];
  If[OptionValue["Export"],
    (
      fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"}];
      Export[fname, T22Table];
    )
  ];
  Return[T22Table];
);
```

```
ReducedT22infn::usage = "ReducedT22infn[n, SL, SpLp] calculates the
    reduced matrix element of the T22 operator for the f^n
    configuration corresponding to the terms SL and SpLp.
This is done by using equation (4) of ''Judd, BR, HM Crosswhite, and
    Hannah Crosswhite. ''Intra-Atomic Magnetic Interactions for f
    Electrons.'' Physical Review 169, no. 1 (1968): 130.''
";
ReducedT22infn[numE_, SL_, SpLp_] := Module[
  {spin, orbital, t, idx1, idx2, S, L,
  Sp, Lp, cfpSL, cfpSpLp, parentSL,
  parentSpLp, Sb, Lb, Tnkk, phase, Sbp, Lbp},
  (
    {spin, orbital} = {1/2, 3};
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    t = 2;
    cfpSL    = CFP[{numE, SL}];
    cfpSpLp  = CFP[{numE, SpLp}];
    Tnkk = Sum[(
      parentSL   = cfpSL[[idx2, 1]];
      parentSpLp = cfpSpLp[[idx1, 1]];
      {Sb, Lb}   = FindSL[parentSL];
      {Sbp, Lbp} = FindSL[parentSpLp];
      phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
      (
        phase *
        cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
        SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
        SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
        T22Table[{numE - 1, parentSL, parentSpLp}]
      )
    ),
    {idx1, 2, Length[cfpSpLp]},
    {idx2, 2, Length[cfpSL]}
    ];
    Tnkk *= numE / (numE - 2) * Sqrt[TPO[S,Sp,L,Lp]];
    Return[Tnkk];
  )
];
```

```
ReducedT22inf2::usage = "ReducedT22inf2[SL, SpLp] returns the reduced
    matrix element of the scalar component of the double tensor T22
```

```
    for the terms SL, SpLp in f^2.
Data used here for m0, m2, m4 is from Table I of Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. Intra-Atomic Magnetic
    Interactions for f Electrons. Physical Review 169, no. 1 (1968):
    130.
";
ReducedT22inf2[SL_, SpLp_] := Module[
  {statePosition, PsiPsipStates, m0, m2, m4, Tkk2m},
  (
    T22inf2 = <|
    {"3P", "3P"} -> -12 M0 - 24 M2 - 300/11 M4,
    {"3P", "3F"} -> 8/Sqrt[3] (3 M0 + M2 - 100/11 M4),
    {"3F", "3F"} -> 4/3 Sqrt[14] (-M0 + 8 M2 - 200/11 M4),
    {"3F", "3H"} -> 8/3 Sqrt[11/2] (2 M0 - 23/11 M2 - 325/121 M4),
    {"3H", "3H"} -> 4/3 Sqrt[143] (M0 - 34/11 M2 - (1325/1573) M4)
    |>;
    Which[
      MemberQ[Keys[T22inf2],{SL,SpLp}],
        Return[T22inf2[{SL,SpLp}]],
      MemberQ[Keys[T22inf2],{SpLp,SL}],
        Return[T22inf2[{SpLp,SL}]],
      True,
        Return[0]
    ]
  )
];
```

The function GenerateSOOandECSOTable calls the function SOOandECSO over all possible combinations of the arguments $\{n, SL, S'L', J\}$ and uses their values to populate the association SOOandECSOTable. In turn the function SOOandECSO queries the precomputed values of &qn-48 as stored in the association SOOandECSOLSTable.

```
GenerateSOOandECSOTable::usage = "GenerateSOOandECSOTable[nmax]
    generates the reduced matrix elements in the |LSJ> basis for the (
    spin-other-orbit + electrostatically-correlated-spin-orbit)
    operator. It returns an association where the keys are of the form
     {n, SL, SpLp, J}. If the option ''Export'' is set to True then
    the resulting object is saved to the data folder. Since this is a
    scalar operator, there is no MJ dependence. This dependence only
    comes into play when the crystal field contribution is taken into
    account.";
Options[GenerateSOOandECSOTable] = {"Export"->False};
GenerateSOOandECSOTable[nmax_, OptionsPattern[]] := (
  SOOandECSOTable = <||>;
  Do[
    SOOandECSOTable[{numE, SL, SpLp, J}] = (SOOandECSO[numE, SL, SpLp
    , J] /. Prescaling);,
  {numE, 1, nmax},
  {J, MinJ[numE], MaxJ[numE]},
  {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
  {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
  ];
  If[OptionValue["Export"],
  (
    fname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
    Export[fname, SOOandECSOTable];
  )
  ];
  Return[SOOandECSOTable];
);
```

```
SOOandECSO::usage = "SOOandECSO[n, SL, SpLp, J] returns the matrix
    element <|SL,J|spin-spin|SpLp,J|> for the combined effects of the
    spin-other-orbit interaction and the electrostatically-correlated-
    spin-orbit (which originates from configuration interaction
    effects) within the configuration f^n. This matrix element is
    independent of MJ. This is obtained by querying the relevant
    reduced matrix element by querying the association
    SOOandECSOLSTable and putting in the adequate phase and 6-j symbol
    . The SOOandECSOLSTable puts together the reduced matrix elements
    from three operators.
This is calculated according to equation (3) in ''Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic
    Interactions for f Electrons.'' Physical Review 169, no. 1 (1968):
     130.''.
";
```

```
SOOandECSO[numE_, SL_, SpLp_, J_] := Module[
  {S, Sp, L, Lp, α, val},
  (
    α = 1;
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            SOOandECSOLSTable[{numE, SL, SpLp}]
          );
    Return[val];
  )
];
```

```
SOOandECSO::usage = "SOOandECSO[n, SL, SpLp, J] returns the matrix
    element <|SL,J|spin-spin|SpLp,J|> for the combined effects of the
    spin-other-orbit interaction and the electrostatically-correlated-
    spin-orbit (which originates from configuration interaction
    effects) within the configuration f^n. This matrix element is
    independent of MJ. This is obtained by querying the relevant
    reduced matrix element by querying the association
    SOOandECSOLSTable and putting in the adequate phase and 6-j symbol
    . The SOOandECSOLSTable puts together the reduced matrix elements
    from three operators.
This is calculated according to equation (3) in ''Judd, BR, HM
    Crosswhite, and Hannah Crosswhite. ''Intra-Atomic Magnetic
    Interactions for f Electrons.'' Physical Review 169, no. 1 (1968):
     130.''.
";
SOOandECSO[numE_, SL_, SpLp_, J_] := Module[
  {S, Sp, L, Lp, α, val},
  (
    α = 1;
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            SOOandECSOLSTable[{numE, SL, SpLp}]
          );
    Return[val];
  )
];
```

The association `SOOandECSOLSTable` is computed by the function `GenerateSOOandECSOLSTable`. This function populates `SOOandECSOLSTable` with keys of the form $\{n, SL, S'L'\}$. It does this by using the function `ReducedSOOandECSOinf2` in the base case of $\underline{f}^2$, and `ReducedSOOandECSOinfn` for configurations above $\underline{f}^2$. When `ReducedSOOandECSOinfn` is called the sum in &qn-41 is carried out using $t = 1$. When `ReducedSOOandECSOinf2` is called the reduced matrix elements from [JCC68] are used.

```
ReducedSOOandECSOinfn::usage = "ReducedSOOandECSOinfn[numE, SL, SpLp]
     calculates the reduced matrix elements of the (spin-other-orbit +
     ECSO) operator for the f^numE configuration corresponding to the
    terms SL and SpLp. This is done recursively, starting from
    tabulated values for f^2 from ''Judd, BR, HM Crosswhite, and
    Hannah Crosswhite. ''Intra-Atomic Magnetic Interactions for f
    Electrons.'' Physical Review 169, no. 1 (1968): 130.'', and by
    using equation (4) of that same paper.
";
ReducedSOOandECSOinfn[numE_, SL_, SpLp_] := Module[
  {spin, orbital, t, S, L, Sp, Lp,
  idx1, idx2, cfpSL, cfpSpLp, parentSL,
  Sb, Lb, Sbp, Lbp, parentSpLp, funval},
  (
    {spin, orbital} = {1/2, 3};
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    t = 1;
    cfpSL    = CFP[{numE, SL}];
    cfpSpLp  = CFP[{numE, SpLp}];
    funval   = Sum[
      (
        parentSL = cfpSL[[idx2, 1]];
```

```
        parentSpLp = cfpSpLp[[idx1, 1]];
        {Sb, Lb}   = FindSL[parentSL];
        {Sbp, Lbp} = FindSL[parentSpLp];
        phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
        (
          phase *
          cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
          SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
          SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
          SOOandECSOLSTable[{numE - 1, parentSL, parentSpLp}]
        )
      ),
    {idx1, 2, Length[cfpSpLp]},
    {idx2, 2, Length[cfpSL]}
    ];
    funval *= numE / (numE - 2) * Sqrt[TPO[S, Sp, L, Lp]];
    Return[funval];
  )
];
```

```
ReducedSOOandECSOinf2::usage = "ReducedSOOandECSOinf2[SL, SpLp]
    returns the reduced matrix element corresponding to the operator (
    T11 + t11 - a13 * z13 / 6) for the terms {SL, SpLp}. This
    combination of operators corresponds to the spin-other-orbit plus
    ECSO interaction.
The T11 operator corresponds to the spin-other-orbit interaction, and
     the t11 operator (associated with electrostatically-correlated
    spin-orbit) originates from configuration interaction analysis. To
     their sum a factor proportional to the operator z13 is subtracted
     since its effect is redundant to the spin-orbit interaction. The
    factor of 1/6 is not on Judd's 1968 paper, but it is on ''Chen,
    Xueyuan, Guokui Liu, Jean Margerie, and Michael F Reid. ''A Few
    Mistakes in Widely Used Data Files for Fn Configurations
    Calculations.'' Journal of Luminescence 128, no. 3 (2008):
    421-27''.
The values for the reduced matrix elements of z13 are obtained from
    Table IX of the same paper. The value for a13 is from table VIII.
Rigorously speaking the Pk parameters here are subscripted. The
    conversion to superscripted parameters is performed elsewhere with
     the Prescaling replacement rules.
";
ReducedSOOandECSOinf2[SL_, SpLp_]  := Module[
  {a13, z13, z13inf2, matElement, redSOOandECSOinf2},
  (
    a13 = (-33 M0 + 3 M2 + 15/11 M4 -
          6 P0 + 3/2 (35 P2 + 77 P4 + 143 P6));
    z13inf2 = <|
        {"1S","3P"} -> 2,
        {"3P","3P"} -> 1,
        {"3P","1D"} -> -Sqrt[(15/2)],
        {"1D","3F"} -> Sqrt[10],
        {"3F","3F"} -> Sqrt[14],
        {"3F","1G"} -> -Sqrt[11],
        {"1G","3H"} -> Sqrt[10],
        {"3H","3H"} -> Sqrt[55],
        {"3H","1I"} -> -Sqrt[(13/2)]
        |>;
    matElement = Which[
      MemberQ[Keys[z13inf2],{SL,SpLp}],
        z13inf2[{SL,SpLp}],
      MemberQ[Keys[z13inf2],{SpLp,SL}],
        z13inf2[{SpLp,SL}],
      True,
        0
    ];
    redSOOandECSOinf2 = (
        ReducedT11inf2[SL, SpLp] +
        Reducedt11inf2[SL, SpLp] -
        a13 / 6 * matElement
    );
    redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
    Return[redSOOandECSOinf2];
  )
];
```

3.8 $\hat{\mathcal{H}}_{\text{人}}$: three-body effective operators

The three-body operators in the semi-empirical Hamiltonian are due to the *configuration-interaction* effects of the Coulomb repulsion. More specifically, they originate from configuration interaction between the ground configuration $(4\underline{f})^n$ and single electron excitations to the $(4\underline{f})^{n\pm1}(n'\underline{\ell}')^{\mp1}$ configurations.

The operators that can be used to span the resulting effects were initially studied by Wybourne and Rajnak in 1963 [RW63], their analysis was complemented soon after by Judd [Jud66], and revisited again by Judd in 1984 [JS84].

This model interaction is spanned by a set of 14 $\hat{t}_i$ of operators ($\hat{t}$ from *t*hree)

$$\hat{\mathcal{H}}_{\text{人}} = T'^{(2)}\hat{t}_2{}' + T'^{(11)}\hat{t}_{11}{}' \sum_{\substack{k=2,3,4,6,7,8,\\ 11,12,14,15,\\ 16,17,18,19}} T^{(k)}\hat{t}_k, \tag{50}$$

where $\hat{t}_2$ and $\hat{t}_{11}$ are operators that have orthogonal alternatives represented by $\hat{t}_2{}'$ and $\hat{t}_{11}{}'$ (see [JS84]). **qlanth** includes the legacy operator $\hat{t}_2$ since it was used for important work during and before the 1980s.

The omission of some indices in this sum has to do with the fact that the way in which these are defined in terms of their index (see [Jud66]) gives rise to two-body operators which can be absorbed by the two-body terms in the Hamiltonian. As such, it is not so much that they are not included, but rather that their effects are considered to be accounted for elsewhere. This is representative of a common feature of configuration interaction: it gives rise to new intra-configuration operators, but it also contributes to already present operators; this makes it harder to approximate the model parameters *ab-initio*, but is not a practical obstacle for the semi-empirical approach (although it certainly complicates the physical interpretation that each parameter has). Furthermore, it is often the case that the operator set is limited to the subset {2,3,4,6,7,8}; a practice that is justified *post-facto* after seeing that these are sufficient to describe the data.

The calculation of a three body operator matrix elements across the $\underline{f}^n$ configurations is analogous to how a two-body operator is calculated. Except that in this case what is needed are the reduced matrix elements in $\underline{f}^3$ and the equation that is used to propagate these across the other configurations is equation 4 of [Jud66] (here adding the explicit dependence on J and M_J):

$$\langle \underline{f}^n\psi|\hat{t}_i|\underline{f}^n\psi'\rangle = \delta(J,J')\delta(M_J,M_J')\frac{n}{n-3}\sum_{\bar{\psi}\bar{\psi}'}\left(\psi\left\{\right|\bar{\psi}\right)\left(\psi'\left\{\right|\bar{\psi}'\right)\langle \underline{f}^{n-1}\bar{\psi}|\hat{t}_i|\underline{f}^{n-1}\bar{\psi}'\rangle. \tag{51}$$

The sum in this expression runs over the parents in $\underline{f}^{n-1}$ that are common to both the daughter terms ψ and ψ' in $\underline{f}^n$. The equation above yielding LSJMJ matrix elements, and being diagonal in J, M_J as is due to a scalar operator.

In **qlanth** this is all implemented in the function `GenerateThreeBodyTables`. Where the matrix elements in $\underline{f}^3$ are from [JS84], where the data has been digitized in the files `Judd1984-1.csv` and `Judd1984-2.csv`, which are parsed through the function `ParseJudd1984`.

In `GenerateThreeBodyTables` a special case is made for $\hat{t}_2$ and $\hat{t}_{11}$ which are calculated differently beyond the half-filled shell. In the case of the other $\hat{t}_k$ operators, beyond $\underline{f}^7$ the matrix elements simply see a global sign flip, whereas in the case of $\hat{t}_2$ and $\hat{t}_{11}$ the coefficients of fractional parentage beyond $\underline{f}^7$ are used. This yields the unexpected result that in the $\underline{f}^{12}$ configuration, which corresponds to two holes, there is a non-zero three body operator $\hat{t}_2$. This is an arcane result that was corrected by Judd in 1984 [JS84], but which lingered long enough that important work in the 1980s was calculated with it. When calculations are carried out, if $\hat{t}_2{}'/\hat{t}_{11}{}'$ is used then $\hat{t}_2/\hat{t}_{11}$ should not be used and vice versa.

One additional feature of $\hat{t}_2$ that needs to be accounted for, is that it doesn't have the simple relationship for conjugate configurations that all the other $\hat{t}_i$ operators have. For the sake of simplicity, and to avoid having to explicitly store matrix elements beyond $\underline{f}^7$ **qlanth** takes the approach of adding a control parameter `t2Switch` which needs to be set to 1 if below or at $\underline{f}^7$ and set to 0 if above $\underline{f}^7$.

```
1 GenerateThreeBodyTables::usage = "This function generates the reduced
      matrix elements for the three body operators using the
     coefficients of fractional parentage, including those beyond f^7."
     ;
2 Options[GenerateThreeBodyTables] = {"Export" -> False};
3 GenerateThreeBodyTables[OptionsPattern[]] := (
```

```
tiKeys         = (StringReplace[ToString[#], {"T" -> "t_{", "p" -> "
  }^{'"}] <> "}") & /@ TSymbols;
TSymbolsAssoc = AssociationThread[tiKeys -> TSymbols];
juddOperators = ParseJudd1984[];
(* op3MatrixElement[SL, SpLp, opSymbol] returns the value for the
  reduced matrix element of the operator opSymbol for the terms {SL,
   SpLp} in the f^3 configuration. *)
op3MatrixElement[SL_, SpLp_, opSymbol_] := (
  jOP = juddOperators[{3, opSymbol}];
  key = {SL, SpLp};
  val = If[MemberQ[Keys[jOP], key],
    jOP[key],
    0];
  Return[val];
);
(* ti: This is the implementation of formula (2) in Judd & Suskin
  1984. It computes the reduced matrix elements of ti in f^n by
  using the reduced matrix elements in f^3 and the coefficients of
  fractional parentage. If the option ''Fast'' is set to True then
  the values for n>7 are simply computed as the negatives of the
  values in the complementary configuration; this except for t2 and
  t11 which are treated as special cases. *)
Options[ti] = {"Fast" -> True};
ti[nE_, SL_, SpLp_, tiKey_, opOrder_ : 3, OptionsPattern[]] :=
  Module[
  {nn, S, L, Sp, Lp,
  cfpSL, cfpSpLp,
  parentSL, parentSpLp,
  tnk, tnks},
  (
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    fast     = OptionValue["Fast"];
    numH     = 14 - nE;
    If[fast && Not[MemberQ[{"t_{2}","t_{11}"},tiKey]] && nE > 7,
      Return[-tktable[{numH, SL, SpLp, tiKey}]]
    ];
    If[(S == Sp && L == Lp),
    (
      cfpSL   = CFP[{nE, SL}];
      cfpSpLp = CFP[{nE, SpLp}];
      tnks = Table[(
          parentSL   = cfpSL[[nn, 1]];
          parentSpLp = cfpSpLp[[mm, 1]];
          cfpSL[[nn, 2]] * cfpSpLp[[mm, 2]] *
          tktable[{nE - 1, parentSL, parentSpLp, tiKey}]
          ),
          {nn, 2, Length[cfpSL]},
          {mm, 2, Length[cfpSpLp]}
          ];
      tnk = Total[Flatten[tnks]];
      ),
    tnk = 0;
    ];
    Return[ nE / (nE - opOrder) * tnk];
  )
];
(* Calculate the reduced matrix elements of t^i for n up to 14 *)
tktable = <||>;
Do[(
  Do[(
    tkValue = Which[numE <= 2,
      (*Initialize n=1,2 with zeros*)
      0,
      numE == 3,
      (* Grab matrix elem in f^3 from Judd 1984 *)
      SimplifyFun[op3MatrixElement[SL, SpLp, opKey]],
      True,
      SimplifyFun[ti[numE, SL, SpLp, opKey, If[opKey == "e_{3}", 2,
   3]]]
      ];
    tktable[{numE, SL, SpLp, opKey}] = tkValue;
    ),
  {SL, AllowedNKSLTerms[numE]},
  {SpLp, AllowedNKSLTerms[numE]},
  {opKey, Append[tiKeys, "e_{3}"]}
```

```
    ];
    PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], "
    configuration complete"]];
    ),
    {numE, 1, 14}
  ];

  (* Now use those reduced matrix elements to determine their sum as
    weighted by their corresponding strengths Ti *)
  ThreeBodyTable = <||>;
  Do[
    Do[
    (
      ThreeBodyTable[{numE, SL, SpLp}] = (
        Sum[(
          If[tiKey == "t_{2}", t2Switch, 1] *
          tktable[{numE, SL, SpLp, tiKey}] *
          TSymbolsAssoc[tiKey] +
          If[tiKey == "t_{2}", 1 - t2Switch, 0] *
          (-tktable[{14 - numE, SL, SpLp, tiKey}]) *
          TSymbolsAssoc[tiKey]
          ),
        {tiKey, tiKeys}
        ]
      );
    ),
    {SL, AllowedNKSLTerms[numE]},
    {SpLp, AllowedNKSLTerms[numE]}
    ];
  PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], " matrix
    complete"]];,
  {numE, 1, 7}
  ];

  ThreeBodyTables = Table[(
    terms = AllowedNKSLTerms[numE];
    singleThreeBodyTable =
      Table[
        {SL, SLp} -> ThreeBodyTable[{numE, SL, SLp}],
        {SL, terms},
        {SLp, terms}
      ];
    singleThreeBodyTable  = Flatten[singleThreeBodyTable];
    singleThreeBodyTables = Table[(
        notNullPosition = Position[TSymbols, notNullSymbol][[1, 1]];
        reps = ConstantArray[0, Length[TSymbols]];
        reps[[notNullPosition]] = 1;
        rep = AssociationThread[TSymbols -> reps];
        notNullSymbol -> Association[(singleThreeBodyTable /. rep)]
        ),
      {notNullSymbol, TSymbols}
      ];
    singleThreeBodyTables = Association[singleThreeBodyTables];
    numE -> singleThreeBodyTables),
    {numE, 1, 7}
];

ThreeBodyTables = Association[ThreeBodyTables];
If[OptionValue["Export"],
  (
    threeBodyTablefname = FileNameJoin[{moduleDir, "data", "
    ThreeBodyTable.m"}];
    Export[threeBodyTablefname, ThreeBodyTable];
    threeBodyTablesfname = FileNameJoin[{moduleDir, "data", "
    ThreeBodyTables.m"}];
    Export[threeBodyTablesfname, ThreeBodyTables];
  )
 ];
Return[{ThreeBodyTable, ThreeBodyTables}];
);
```

```
ParseJudd1984::usage = "This function parses the data from tables 1
    and 2 of Judd from Judd, BR, and MA Suskin. ''Complete Set of
    Orthogonal Scalar Operators for the Configuration f^3''. JOSA B 1,
     no. 2 (1984): 261-65.";
Options[ParseJudd1984] = {"Export" -> False};
```

```
ParseJudd1984[OptionsPattern[]] := (
  ParseJuddTab1[str_] := (
    strR = ToString[str];
    strR = StringReplace[strR, ".5" -> "^(1/2)"];
    num = ToExpression[strR];
    sign = Sign[num];
    num = sign*Simplify[Sqrt[num^2]];
    If[Round[num] == num, num = Round[num]];
    Return[num]);

  (* Parse table 1 from Judd 1984 *)
  judd1984Fname1 = FileNameJoin[{moduleDir, "data", "Judd1984-1.csv"}];
  data = Import[judd1984Fname1, "CSV", "Numeric" -> False];
  headers = data[[1]];
  data = data[[2 ;;]];
  data = Transpose[data];
  \[Psi] = Select[data[[1]], # != "" &];
  \[Psi]p = Select[data[[2]], # != "" &];
  matrixKeys = Transpose[{\[Psi], \[Psi]p}];
  data = data[[3 ;;]];
  cols = Table[ParseJuddTab1 /@ Select[col, # != "" &], {col, data}];
  cols = Select[cols, Length[#] == 21 &];
  tab1 = Prepend[Prepend[cols, \[Psi]p], \[Psi]];
  tab1 = Transpose[Prepend[Transpose[tab1], headers]];

  (* Parse table 2 from Judd 1984 *)
  judd1984Fname2 = FileNameJoin[{moduleDir, "data", "Judd1984-2.csv"}];
  data = Import[judd1984Fname2, "CSV", "Numeric" -> False];
  headers = data[[1]];
  data = data[[2 ;;]];
  data = Transpose[data];
  {operatorLabels, WUlabels, multiFactorSymbols, multiFactorValues} =
     data[[;; 4]];
  multiFactorValues = ParseJuddTab1 /@ multiFactorValues;
  multiFactorValues = AssociationThread[multiFactorSymbols ->
    multiFactorValues];

  (*scale values of table 1 given the values in table 2*)
  oppyS = {};
  normalTable =
    Table[header = col[[1]];
      If[StringContainsQ[header, " "],
        (
          multiplierSymbol = StringSplit[header, " "][[1]];
          multiplierValue = multiFactorValues[multiplierSymbol];
          operatorSymbol = StringSplit[header, " "][[2]];
          oppyS = Append[oppyS, operatorSymbol];
        ),
        (
          multiplierValue = 1;
          operatorSymbol = header;
        )
      ];
      normalValues = 1/multiplierValue*col[[2 ;;]];
      Join[{operatorSymbol}, normalValues], {col, tab1[[3 ;;]]}
    ];

  (*Create an association for the reduced matrix elements in the f^3
    config*)
  juddOperators = Association[];
  Do[(
    col      = normalTable[[colIndex]];
    opLabel  = col[[1]];
    opValues = col[[2 ;;]];
    opMatrix = AssociationThread[matrixKeys -> opValues];
    Do[(
      opMatrix[Reverse[mKey]] = opMatrix[mKey]
      ),
    {mKey, matrixKeys}
    ];
    juddOperators[{3, opLabel}] = opMatrix),
    {colIndex, 1, Length[normalTable]}
  ];

```

```
  (* special case of t2 in f3 *)
  (* this is the same as getting the reduced matrix elements from
    Judd 1966 *)
  numE = 3;
  e3Op       = juddOperators[{3, "e_{3}"}];
  t2prime    = juddOperators[{3, "t_{2}^{'}"}];
  prefactor = 1/(70 Sqrt[2]);
  t2Op = (# -> (t2prime[#] + prefactor*e3Op[#])) & /@ Keys[t2prime];
  t2Op = Association[t2Op];
  juddOperators[{3, "t_{2}"}] = t2Op;

  (*Special case of t11 in f3*)
  t11 = juddOperators[{3, "t_{11}"}];
  eβprimeOp = juddOperators[{3, "e_{\\beta}^{'}"}];
  t11primeOp = (# -> (t11[#] + Sqrt[3/385] eβprimeOp[#])) & /@ Keys[
    t11];
  t11primeOp = Association[t11primeOp];
  juddOperators[{3, "t_{11}^{'}"}] = t11primeOp;
  If[OptionValue["Export"],
    (
      (*export them*)
      PrintTemporary["Exporting ..."];
      exportFname = FileNameJoin[{moduleDir, "data", "juddOperators.m
    "}];
      Export[exportFname, juddOperators];
    )
  ];
  Return[juddOperators];
);

ThreeBodyTable::usage="ThreeBodyTable is an association containing
    the LS-reduced matrix elements for the three-body operators for f^
    n configurations. The keys are lists of the form {n, SL, SpLp}.";
```

3.9 $\hat{\mathcal{H}}_{\mathbf{cf}}$: crystal-field

The crystal-field partially accounts for the influence of the surrounding lattice on the ion. The simplest picture of this influence imagines the lattice as responsible for an electric field felt at the position of the ion. This electric field corresponding to an electrostatic potential described as a multipolar sum of the form:

$$V(r_i, \theta_i, \phi_i) = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{A}_q^{(k)} r_i^k \mathcal{C}_q^{(k)}(\theta_i, \phi_i) \tag{52}$$

where the $\mathcal{C}_q^{(k)}$ are spherical harmonics normalized with the Racah convention

$$\mathcal{C}_q^{(k)} = \sqrt{\frac{4\pi}{2k+1}} Y_q^{(k)}. \tag{53}$$

Here we have chosen a coordinate system with its origin at the position of the nucleus, and in which we only have positive powers of the distance r_i because we have expanded the contributions from all the surrounding ions as a sum over spherical harmonics centered at the position of the nucleus, without r ever large enough to reach any of the positions of the lattice ions.

Furthermore, since we have n valence electrons, then the total crystal field potential is

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=-k}^{k} \mathcal{A}_q^{(k)} r_i^k \mathcal{C}_q^{(k)}(\theta_i, \phi_i). \tag{54}$$

And if we average the radial coordinate,

$$\hat{\mathcal{H}}_{\mathrm{cf}} = \sum_{i=1}^{n} \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{B}_q^{(k)} \mathcal{C}_q^{(k)}(i) \tag{55}$$

where the radial average is included as

$$\mathcal{B}_q^{(k)} := \mathcal{A}_q^{(k)} \langle 4f | r^k | 4f \rangle. \tag{56}$$

$\mathcal{B}_q^{(k)}$ may be complex in general. However, since the sum in Eqn-54 needs to result in a real and Hermitian operator, there are restrictions on $\mathcal{B}_q^{(k)}$ that need to be accounted for. Once the behavior of $\mathcal{C}_q^{(k)}$ under complex conjugation is considered, $\mathcal{C}_q^{(k)*} = (-1)^q \mathcal{C}_{-q}^{(k)}$, it is necessary that

$$\mathcal{B}_q^{(k)} = (-1)^q \mathcal{B}_{-q}^{(k)*}. \tag{57}$$

Presently the sum over q spans both its negative and positive values. This can be limited to only the non-negative values of q. Separating the real and imaginary parts of $\mathcal{B}_q^{(k)}$ such that $\mathcal{B}_q^{(k)} = B_q^{(k)} + iS_q^{(k)}$ for $q \neq 0$ and $\mathcal{B}_0^{(k)} = 2B_0^{(k)}$ the sum for the crystal field can then be written as

$$\hat{\mathcal{H}}_{\text{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=0}^{k} B_q^{(k)} \left(\mathcal{C}_q^{(k)} + (-1)^q \mathcal{C}_{-q}^{(k)} \right) + iS_q^{(k)} \left(\mathcal{C}_q^{(k)} - (-1)^q \mathcal{C}_{-q}^{(k)} \right). \tag{58}$$

A staple of the Wigner-Racah algebra is writing up operators of interest in terms of standard ones for which the matrix elements are straightforward. One such operator is the unit tensor operator $\hat{u}^{(k)}$ for a single electron. The Wigner-Eckart theorem – on which all of this algebra is an elaboration – effectively separates the dynamical and geometrical parts of a given interaction; the unit tensor operators isolate the geometric contributions. This irreducible tensor operator $\hat{u}^{(k)}$ is defined as the tensor operator having the following reduced matrix elements (written in terms of the triangular delta, see section on notation):

$$\langle \ell \| \hat{u}^{(k)} \| \ell' \rangle = 1. \tag{59}$$

In terms of this tensor one may then define the symmetric (in the sense that the resulting operator is equitable among all electrons) unit tensor operator for n particles as

$$\hat{U}^{(k)} = \sum_{i}^{n} \hat{u}_i^{(k)}. \tag{60}$$

This tensor is relevant to the calculation of the above matrix elements since

$$\mathcal{C}_q^{(k)} = \langle \underline{\ell} \| \mathcal{C}^{(k)} \| \underline{\ell}' \rangle \hat{u}_q^{(k)} = (-1)^{\underline{\ell}} \sqrt{[\![\underline{\ell}]\!][\![\underline{\ell}']\!]} \begin{pmatrix} \underline{\ell} & k & \underline{\ell}' \\ 0 & 0 & 0 \end{pmatrix} \hat{u}_q^{(k)}. \tag{61}$$

With this, the matrix elements of $\hat{\mathcal{H}}_{\text{cf}}$ in the $|LSJM_J\rangle$ basis are:

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{\mathcal{H}}_{\text{cf}} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-3}} = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{B}_q^{(k)} \langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle \langle \underline{\ell} \| \hat{C}^{(k)} \| \underline{\ell} \rangle \tag{62}$$

where the matrix elements of $\hat{U}_q^{(k)}$ can be resolved with a 3j symbol as

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' S'L'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-4}} = (-1)^{J-M_J} \begin{pmatrix} J & k & J' \\ -M_J & q & M_{J'} \end{pmatrix} \langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle \tag{63}$$

and reduced a second time with the inclusion of a 6j symbol resulting in

$$\overbrace{\langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Wybourne eqn. 6-5}} = (-1)^{S+L+J'+k} \sqrt{[\![J]\!][\![J']\!]} \times \begin{Bmatrix} J & J' & k \\ L' & L & S \end{Bmatrix} \langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle. \tag{64}$$

This last reduced matrix element is finally computed by summing over $\bar{\alpha}\bar{L}\bar{S}$ which are the $\underline{\ell}^{n-1}$ parents which are common to $|\alpha LS\rangle$ and $|\alpha' L'S'\rangle$ from the $\underline{\ell}^n$ configuration:

$$\overbrace{\langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Cowan eqn. 11.53}} = \delta(S,S') n (-1)^{\underline{\ell}+L+k} \sqrt{[\![L]\!][\![L']\!]} \times \sum_{\bar{\alpha}\bar{L}\bar{S}} (-1)^{\bar{L}} \begin{Bmatrix} \underline{\ell} & k & \underline{\ell} \\ L & \bar{L} & L' \end{Bmatrix} \left(\underline{\ell}^n \alpha LS \left\{ \underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} \right) \left(\underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} \right\} \underline{\ell}^n \alpha' L'S' \right). \tag{65}$$

From the $\langle \underline{\ell} \| \hat{C}^{(k)} \| \underline{\ell} \rangle$, and given that we are using $\underline{\ell} = \underline{\mathrm{f}} = 3$, we can see that by the triangular condition $\triangleleft(3, k, 3)$ the non-zero contributions only come from $k = 0, 1, 2, 3, 4, 5, 6$. An additional selection rule on k comes from considerations of parity. Since both the bra and the ket in $\langle \underline{\ell}^n \alpha S L J M_J | \hat{\mathcal{H}}_{\mathrm{cf}} | \underline{\ell}^n \alpha' S L' J' M_{J'} \rangle$ have the same parity, then the overall parity of the braket is determined by the parity of $\mathcal{C}_q^{(k)}$, and since the parity of $\mathcal{C}_q^{(k)}$ is $(-1)^k$ then for the braket to be non-zero we require that k should also be even. In view of this, in all the above equations for the crystal field the values for k should be limited to $2, 4, 6$. The value of $k = 0$ having been omitted from the start since this only contributes a common energy shift. Putting everything together:

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=2,4,6} \sum_{q=0}^{k} B_q^{(k)} \left(\mathcal{C}_q^{(k)} + (-1)^q \mathcal{C}_{-q}^{(k)} \right) + i S_q^{(k)} \left(\mathcal{C}_q^{(k)} - (-1)^q \mathcal{C}_{-q}^{(k)} \right). \tag{66}$$

The above equations are implemented in **qlanth** by the function `CrystalField`. This function puts together the symbolic sum in Eqn-62 by using the function `Cqk`. `Cqk` then uses the diagonal reduced matrix elements of $\mathcal{C}_q^{(k)}$ and the precomputed values for `Uk` (stored in `ReducedUkTable`).

The required reduced matrix elements of $\hat{U}^{(k)}$ are calculated by the function `ReducedUk`, which is used by `GenerateReducedUkTable` to precompute its values.

```
Bqk::usage = "Real part of the Bqk coefficients.";
Bqk[q_, 2] := {B02/2, B12, B22}[[q + 1]];
Bqk[q_, 4] := {B04/2, B14, B24, B34, B44}[[q + 1]];
Bqk[q_, 6] := {B06/2, B16, B26, B36, B46, B56, B66}[[q + 1]];
```

```
Sqk::usage = "Imaginary part of the Bqk coefficients.";
Sqk[q_, 2] := {0, S12, S22}[[q + 1]];
Sqk[q_, 4] := {0, S14, S24, S34, S44}[[q + 1]];
Sqk[q_, 6] := {0, S16, S26, S36, S46, S56, S66}[[q + 1]];
```

```
Cqk::usage = "Cqk[numE, q, k, NKSL, J, M, NKSLp, Jp, Mp]. In Wybourne
    (1965) see equations 6-3, 6-4, and 6-5. Also in TASS see equation
    11.53.";
Cqk[numE_, q_, k_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_] := Module[
  {S, Sp, L, Lp, orbital, val},
  (
    orbital = 3;
    {S, L}   = FindSL[NKSL];
    {Sp, Lp} = FindSL[NKSLp];
    f1  = ThreeJay[{J, -M}, {k, q}, {Jp, Mp}];
    val =
      If[f1==0,
        0,
        (
          f2 = SixJay[{L, J, S}, {Jp, Lp, k}] ;
          If[f2==0,
            0,
            (
              f3 = ReducedUkTable[{numE, orbital, NKSL, NKSLp, k}];
              If[f3==0,
                0,
                (
                  (
                    Phaser[J - M + S + Lp + J + k] *
                    Sqrt[TPO[J, Jp]] *
                    f1 *
                    f2 *
                    f3 *
                    Ck[orbital, k]
                  )
                )
              ]
            )
          ]
        )
      ];
    Return[val];
  )
];
```

```
CrystalField::usage = "CrystalField[n, NKSL, J, M, NKSLp, Jp, Mp]
    calculates the matrix element of the crystal field in terms of Bqk
     and Sqk parameters for configuration f^numE. It is calculated as
    an association with keys of the form {n, NKSL, J, M, NKSLp, Jp, Mp
    }.";
CrystalField[numE_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_] := (
  Sum[
    (
      cqk  = Cqk[numE,  q, k, NKSL, J, M, NKSLp, Jp, Mp];
      cmqk = Cqk[numE, -q, k, NKSL, J, M, NKSLp, Jp, Mp];
      Bqk[q, k]   * (cqk + (-1)^q * cmqk) +
      I*Sqk[q, k] * (cqk - (-1)^q * cmqk)
      ),
  {k, {2, 4, 6}},
  {q, 0, k}
  ]
)
```

```
ReducedUk::usage = "ReducedUk[n, l, SL, SpLp, k] gives the reduced
    matrix element of the symmetric unit tensor operator U^(k). See
    equation 11.53 in TASS.";
ReducedUk[numE_, l_, SL_, SpLp_, k_] := Module[
  {spin, orbital, Uk, S, L,
  Sp, Lp, Sb, Lb, parentSL,
  cfpSL, cfpSpLp, Ukval,
  SLparents, SLpparents,
  commonParents, phase},
    {spin, orbital} = {1/2, 3};
    {S, L}          = FindSL[SL];
    {Sp, Lp}        = FindSL[SpLp];
    If[Not[S == Sp],
      Return[0]
    ];
    cfpSL       = CFP[{numE, SL}];
    cfpSpLp     = CFP[{numE, SpLp}];
    SLparents   = First /@ Rest[cfpSL];
    SLpparents  = First /@ Rest[cfpSpLp];
    commonParents = Intersection[SLparents, SLpparents];
    Uk = Sum[(
      {Sb, Lb} = FindSL[\[Psi]b];
      Phaser[Lb] *
        CFPAssoc[{numE, SL, \[Psi]b}] *
        CFPAssoc[{numE, SpLp, \[Psi]b}] *
        SixJay[{orbital, k, orbital}, {L, Lb, Lp}]
    ),
    {\[Psi]b, commonParents}
    ];
    phase     = Phaser[orbital + L + k];
    prefactor = numE * phase * Sqrt[TPO[L,Lp]];
    Ukval     = prefactor*Uk;
    Return[Ukval];
]
```

Each of the 32 crystallographic point groups requires only a limited number of non-zero crystal field parameters. In **qlanth** these can be queried programmatically with the use of the function `CrystalFieldForm`. These were taken from a table in Benelli and Gatteschi [BG15] and their corresponding expressions (for a single electron) are in the equations below with a table linking to the corresponding equations. Note that these expressions bring with them an implicit choice for the orientation of the coordinate system (see Section 4).

$\mathcal{S}_2$: Eqn-67	$\mathcal{C}_s$: Eqn-68	$\mathcal{C}_{1h}$: Eqn-69	$\mathcal{C}_2$: Eqn-70	$\mathcal{C}_{2h}$: Eqn-71
$\mathcal{C}_{2v}$: Eqn-72	$\mathcal{D}_2$: Eqn-73	$\mathcal{D}_{2h}$: Eqn-74	$\mathcal{S}_4$: Eqn-75	$\mathcal{C}_4$: Eqn-76
$\mathcal{C}_{4h}$: Eqn-77	$\mathcal{D}_{2d}$: Eqn-78	$\mathcal{C}_{4v}$: Eqn-79	$\mathcal{D}_4$: Eqn-80	$\mathcal{D}_{4h}$: Eqn-81
$\mathcal{C}_3$: Eqn-82	$\mathcal{S}_6$: Eqn-83	$\mathcal{C}_{3h}$: Eqn-84	$\mathcal{C}_{3v}$: Eqn-85	$\mathcal{D}_3$: Eqn-86
$\mathcal{D}_{3d}$: Eqn-87	$\mathcal{D}_{3h}$: Eqn-88	$\mathcal{C}_6$: Eqn-89	$\mathcal{C}_{6h}$: Eqn-90	$\mathcal{C}_{6v}$: Eqn-91
$\mathcal{D}_6$: Eqn-92	$\mathcal{D}_{6h}$: Eqn-93	$\mathcal{T}$: Eqn-94	$\mathcal{T}_h$: Eqn-95	$\mathcal{T}_d$: Eqn-96
$\mathcal{O}$: Eqn-97	$\mathcal{O}_h$: Eqn-98			

Table 1: Expressions for the crystal field in the 32 crystallographic point groups

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{S}_2\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_1^{(2)}\mathcal{C}_1^{(2)} + \left(B_2^{(2)} + iS_2^{(2)}\right)\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)}$$
$$+ \left(B_1^{(4)} + iS_1^{(4)}\right)\mathcal{C}_1^{(4)} + \left(B_2^{(4)} + iS_2^{(4)}\right)\mathcal{C}_2^{(4)} + \left(B_3^{(4)} + iS_3^{(4)}\right)\mathcal{C}_3^{(4)} + \left(B_4^{(4)} + iS_4^{(4)}\right)\mathcal{C}_4^{(4)}$$
$$+ B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_1^{(6)} + iS_1^{(6)}\right)\mathcal{C}_1^{(6)} + \left(B_2^{(6)} + iS_2^{(6)}\right)\mathcal{C}_2^{(6)} + \left(B_3^{(6)} + iS_3^{(6)}\right)\mathcal{C}_3^{(6)}$$
$$+ \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)} + \left(B_5^{(6)} + iS_5^{(6)}\right)\mathcal{C}_5^{(6)} + \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)} \quad (67)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_s\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + \left(B_2^{(4)} + iS_2^{(4)}\right)\mathcal{C}_2^{(4)}$$
$$+ \left(B_4^{(4)} + iS_4^{(4)}\right)\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_2^{(6)} + iS_2^{(6)}\right)\mathcal{C}_2^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)}$$
$$+ \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)} \quad (68)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_{1h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + \left(B_2^{(4)} + iS_2^{(4)}\right)\mathcal{C}_2^{(4)}$$
$$+ \left(B_4^{(4)} + iS_4^{(4)}\right)\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_2^{(6)} + iS_2^{(6)}\right)\mathcal{C}_2^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)}$$
$$+ \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)} \quad (69)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_2\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + \left(B_2^{(4)} + iS_2^{(4)}\right)\mathcal{C}_2^{(4)}$$
$$+ \left(B_4^{(4)} + iS_4^{(4)}\right)\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_2^{(6)} + iS_2^{(6)}\right)\mathcal{C}_2^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)}$$
$$+ \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)} \quad (70)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_{2h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + \left(B_2^{(4)} + iS_2^{(4)}\right)\mathcal{C}_2^{(4)}$$
$$+ \left(B_4^{(4)} + iS_4^{(4)}\right)\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_2^{(6)} + iS_2^{(6)}\right)\mathcal{C}_2^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)}$$
$$+ \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)} \quad (71)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_{2v}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_2^{(4)}\mathcal{C}_2^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)}$$
$$+ B_0^{(6)}\mathcal{C}_0^{(6)} + B_2^{(6)}\mathcal{C}_2^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \quad (72)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{D}_2\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_2^{(4)}\mathcal{C}_2^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)}$$
$$+ B_0^{(6)}\mathcal{C}_0^{(6)} + B_2^{(6)}\mathcal{C}_2^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \quad (73)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{D}_{2h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_2^{(2)}\mathcal{C}_2^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_2^{(4)}\mathcal{C}_2^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)}$$
$$+ B_0^{(6)}\mathcal{C}_0^{(6)} + B_2^{(6)}\mathcal{C}_2^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \quad (74)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{S}_4\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)} \quad (75)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_4\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)} \quad (76)$$

$$\hat{\mathscr{H}}_{\rm cf}\left(\mathcal{C}_{4h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + \left(B_4^{(6)} + iS_4^{(6)}\right)\mathcal{C}_4^{(6)} \quad (77)$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{2d}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{78}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{4v}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{79}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{4}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{80}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{4h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{81}$$

$$\begin{aligned}\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{3}\right) = {} & B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_3^{(4)}\mathcal{C}_3^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} \\ & + \left(B_3^{(6)} + iS_3^{(6)}\right)\mathcal{C}_3^{(6)} + \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)}\end{aligned} \tag{82}$$

$$\begin{aligned}\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{S}_{6}\right) = {} & B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_3^{(4)}\mathcal{C}_3^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} \\ & + \left(B_3^{(6)} + iS_3^{(6)}\right)\mathcal{C}_3^{(6)} + \left(B_6^{(6)} + iS_6^{(6)}\right)\mathcal{C}_6^{(6)}\end{aligned} \tag{83}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{3h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{84}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{3v}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_3^{(4)}\mathcal{C}_3^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_3^{(6)}\mathcal{C}_3^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{85}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{3}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_3^{(4)}\mathcal{C}_3^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_3^{(6)}\mathcal{C}_3^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{86}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{3d}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_3^{(4)}\mathcal{C}_3^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_3^{(6)}\mathcal{C}_3^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{87}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{3h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{88}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{6}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{89}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{6h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{90}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{C}_{6v}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{91}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{6}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{92}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{D}_{6h}\right) = B_0^{(2)}\mathcal{C}_0^{(2)} + B_0^{(4)}\mathcal{C}_0^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_6^{(6)}\mathcal{C}_6^{(6)} \tag{93}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{T}\right) = B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{94}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{T}_{h}\right) = B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{95}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{T}_{d}\right) = B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{96}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{O}\right) = B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{97}$$

$$\hat{\mathcal{H}}_{\rm cf}\left(\mathcal{O}_{h}\right) = B_0^{(4)}\mathcal{C}_0^{(4)} + B_4^{(4)}\mathcal{C}_4^{(4)} + B_0^{(6)}\mathcal{C}_0^{(6)} + B_4^{(6)}\mathcal{C}_4^{(6)} \tag{98}$$

(END) Crystal field expressions (END)

```
CrystalFieldForm::usage = "CrystalFieldForm[symmetryGroup] returns an
     association that describes the crystal field parameters that are
    necessary to describe a crystal field for the given symmetry group
    .

The symmetry group must be given as a string in Schoenflies notation
    and must be one of C1, Ci, S2, Cs, C1h, C2, C2h, C2v, D2, D2h, S4,
     C4, C4h, D2d, C4v, D4, D4h, C3, S6, C3h, C3v, D3, D3d, D3h, C6,
    C6h, C6v, D6, D6h, T, Th, Td, O, Oh.

The returned association has three keys:
  ''BqkSqk'' whose values is a list with the nonzero Bqk and Sqk
    parameters;
  ''constraints'' whose value is either an empty list, or a lists of
    replacements rules that are constraints on the Bqk and Sqk
    parameters;
  ''simplifier'' whose value is an association that can be used to
    set to zero the crystal field parameters that are zero for the
    given symmetry group;
  ''aliases'' whose value is a list with the integer by which the
    point group is also known for and an alternate Schoenflies symbol
    if it exists.

This uses data from table 3.3 in Benelli and Gatteschi, 2015.";
CrystalFieldForm[symmetryGroupString_] := (
  If[Not@ValueQ[crystalFieldFunctionalForms],
    crystalFieldFunctionalForms = Import[FileNameJoin[{moduleDir, "
    data", "crystalFieldFunctionalForms.m"}]];
  ];
  cfForm = crystalFieldFunctionalForms[symmetryGroupString];
  simplifier = Association[(# -> 0) &/@ Complement[cfSymbols, cfForm[
    "BqkSqk"]]];
  Return[Join[cfForm, <|"simplifier"->simplifier|>]];
)
```

3.10 $\hat{\mu}$ and $\hat{\mathscr{H}}_{\mathbf{Z}}$: the magnetic dipole operator and the Zeeman term

In Hartree atomic units, the operator associated with the magnetic dipole operator for an electron is

$$\hat{\mu} = -\mu_{\mathrm{B}} \left(\hat{L} + g_s \hat{S}\right)^{(1)}, \text{with } \mu_{\mathrm{B}} = 1/2. \tag{99}$$

Here we have emphasized the fact that the magnetic dipole operator corresponds to a rank-1 spherical tensor operator.

In the $|LSJM_J\rangle$ basis that we use in **qlanth** the LSJ reduced-matrix elements are computed using equation 15.7 in [Cow81]

$$\langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle = \delta(\alpha LSJ, \alpha' L'S'J') \sqrt{J(J+1)(2J+1)} + \\ \delta(\alpha LS, \alpha' L'S') (-1)^{L+S+J+1} \sqrt{[\![J]\!][\![J]\!]} \begin{Bmatrix} L & S & J \\ 1 & J' & S \end{Bmatrix}. \tag{100}$$

Then these reduced matrix elements are used to resolve the M_J components for $q = -1, 0, 1$ through Wigner-Eckart:

$$\langle \alpha LSJM_J | \left(\hat{L} + g_s \hat{S}\right)^{(1)}_q | \alpha' L'S'J'M_{J'} \rangle = \\ (-1)^{J-M_J} \begin{pmatrix} J & 1 & J' \\ -M_J & q & M'_J \end{pmatrix} \langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle. \tag{101}$$

These two above are put together in `JJBlockMagDip` for given $\{n, J, J'\}$ returning a rank-3 array representing the quantities $\{M_J, M'_J, q\}$.

```
JJBlockMagDip::usage = "JJBlockMagDip[numE, J, Jp] returns an array
    for the LSJM matrix elements of the magnetic dipole operator
    between states with given J and Jp. The option ''Sparse'' can be
    used to return a sparse matrix. The default is to return a sparse
    matrix.
See eqn 15.7 in TASS.
Here it is provided in atomic units in which the Bohr magneton is
    1/2.
```

```
\[Mu] = -(1/2) (L + gs S)
We are using the Racah convention for the reduced matrix elements in
    the Wigner-Eckart theorem. See TASS eqn 11.15.
";
Options[JJBlockMagDip]={"Sparse"->True};
JJBlockMagDip[numE_, braJ_, ketJ_, OptionsPattern[]] := Module[
  {braSLJs, ketSLJs,
  braSLJ,   ketSLJ,
  braSL,    ketSL,
  braS,     braL,
  ketS,     ketL,
  braMJ,    ketMJ,
  matValue, magMatrix,
  summand1, summand2,
  threejays},
  (
    braSLJs  = AllowedNKSLJMforJTerms[numE,braJ];
    ketSLJs  = AllowedNKSLJMforJTerms[numE,ketJ];
    magMatrix  = Table[
      braSL     = braSLJ[[1]];
      ketSL     = ketSLJ[[1]];
      {braS, braL}   = FindSL[braSL];
      {ketS, ketL}   = FindSL[ketSL];
      braMJ       = braSLJ[[3]];
      ketMJ       = ketSLJ[[3]];
      summand1    = If[Or[braJ  != ketJ,
                          braSL != ketSL],
        0,
        Sqrt[braJ*(braJ+1)*TPO[braJ]]
      ];
      (* looking at the string includes checking L=L', S=S', and \
    alpha=\alpha'*)
      summand2 = If[braSL!= ketSL,
        0,
        (gs-1) *
          Phaser[braS+braL+ketJ+1] *
            Sqrt[TPO[braJ]*TPO[ketJ]] *
              SixJay[{braJ,1,ketJ},{braS,braL,braS}] *
                Sqrt[braS(braS+1)TPO[braS]]
      ];
      matValue = summand1 + summand2;
      (* We are using the Racah convention for red matrix elements in
     Wigner-Eckart *)
      threejays = (ThreeJay[{braJ, -braMJ}, {1, #}, {ketJ, ketMJ}] &)
     /@ {-1,0,1};
      threejays *= Phaser[braJ-braMJ];
      matValue  = - 1/2 * threejays * matValue;
      matValue,
    {braSLJ, braSLJs},
    {ketSLJ, ketSLJs}
    ];
    If[OptionValue["Sparse"],
      magMatrix = SparseArray[magMatrix]
    ];
    Return[magMatrix];
  )
];
```

The JJ' blocks that are generated with this function are then put together by `MagDipoleMatrixAssembly` into the final matrix form and the cartesian components calculated according to

$$\hat{\mu}_x = \frac{\hat{\mu}^{(1)}_{-1} - \hat{\mu}^{(1)}_{+1}}{\sqrt{2}}, \tag{102}$$

$$\hat{\mu}_y = i\frac{\hat{\mu}^{(1)}_{-1} + \hat{\mu}^{(1)}_{+1}}{\sqrt{2}}, \tag{103}$$

$$\hat{\mu}_z = \hat{\mu}^{(1)}_0. \tag{104}$$

```
MagDipoleMatrixAssembly::usage = "MagDipoleMatrixAssembly[numE]
    returns the matrix representation of the operator - 1/2 (L + gs S)
     in the f^numE configuration. The function returns a list with
    three elements corresponding to the x,y,z components of this
    operator. The option ''FilenameAppendix'' can be used to append a
```

```
    string to the filename from which the function imports from in
    order to patch together the array. For numE beyond 7 the function
    returns the same as for the complementary configuration. The
    option ''ReturnInBlocks'' can be used to return the matrices in
    blocks. The default is to return the matrices in flattened form
    and as sparse array.";
Options[MagDipoleMatrixAssembly]={
  "FilenameAppendix"->"",
  "ReturnInBlocks"->False};
MagDipoleMatrixAssembly[nf_Integer, OptionsPattern[]] := Module[
  {ImportFun, numE, appendTo,
  emFname, JJBlockMagDipTable,
  Js, howManyJs, blockOp,
  rowIdx, colIdx},
  (
    ImportFun = ImportMZip;
    numE      = nf;
    numH      = 14 - numE;
    numE      = Min[numE,numH];

    appendTo  = (OptionValue["FilenameAppendix"]<>"-magDip");
    emFname   = JJBlockMatrixFileName[numE,"FilenameAppendix"->
    appendTo];
    JJBlockMagDipTable = ImportFun[emFname];

    Js        = AllowedJ[numE];
    howManyJs = Length[Js];
    blockOp   = ConstantArray[0,{howManyJs,howManyJs}];
    Do[
      blockOp[[rowIdx,colIdx]] = JJBlockMagDipTable[{numE,Js[[rowIdx
    ]],Js[[colIdx]]}],
      {rowIdx,1,howManyJs},
      {colIdx,1,howManyJs}
    ];
    If[OptionValue["ReturnInBlocks"],
      (
        opMinus = Map[#[[1]]&, blockOp, {4}];
        opZero  = Map[#[[2]]&, blockOp, {4}];
        opPlus  = Map[#[[3]]&, blockOp, {4}];
        opX =   (opMinus - opPlus)/Sqrt[2];
        opY = I (opPlus + opMinus)/Sqrt[2];
        opZ = opZero;
      ),
        blockOp = ArrayFlatten[blockOp];
        opMinus = blockOp[[;; , ;; , 1]];
        opZero  = blockOp[[;; , ;; , 2]];
        opPlus  = blockOp[[;; , ;; , 3]];
        opX =   (opMinus - opPlus)/Sqrt[2];
        opY = I (opPlus + opMinus)/Sqrt[2];
        opZ = opZero;
    ];
    Return[{opX, opY, opZ}];
  )
];
```

Using the cartesian components of the magnetic dipole operator, the matrix elements of the Zeeman term can then be evaluated. This term can be included in the Hamiltonian through an option in `EffectiveHamiltonian`. Since the magnetic dipole operator is calculated in atomic units, and it seeming desirable that the input units of the magnetic field be Tesla, a conversion factor is included so that the final terms be congruent with the energy units assumed in the other terms in the Hamiltonian, namely the energy pseudo-unit Kayser (cm^{-1}). The conversion factor is called `teslaToKayser` in the file `qonstants.m`.

3.11 Alternative operator bases

One feature from the operators used in &qn-1 is that when data is fit to this model, the parameters are correlated. This has the consequence that using a partial set of operators (say those describing the free-ion part) results in parameter values that change noticeably (by perhaps 10%) when additional interactions are brought into the analysis. The semi-empirical Hamiltonian as written in &qn-1 can be described as a linear combination of a basis set of operators. Correlations in fitted parameters may be removed by using a different operator basis, with this having the added benefit or reducing parameters uncertainties [New82]. This removal of correlations is achieved by making the basis operators

pair-wise orthogonal between themselves.[18]

The operators $\hat{f}_k$, $\hat{L}^2$, $\hat{C}(\mathcal{G}_2)$, $\hat{C}(\mathcal{SO}(7))$, and $\hat{t}_2$ can be made orthogonal among themselves as prescribed by Judd, Crosswhite, and Suskin [JC84; JS84]. In there the change in the operator basis has an accompanying relationship between the coefficients in the old and the new bases, as given below. (Note the n dependence on the equation for $E^{(3)}_\perp$.)

$$E^{(1)}_\perp = \frac{4\alpha}{5} + \frac{\beta}{30} + \frac{\gamma}{25} + \frac{14F^{(2)}}{405} + \frac{7F^{(4)}}{297} + \frac{350F^{(6)}}{11583} \tag{105}$$

$$E^{(2)}_\perp = \frac{F^{(2)}}{2025} - \frac{F^{(4)}}{3267} + \frac{175F^{(6)}}{1656369} \tag{106}$$

$$E^{(3)}_\perp = -\frac{2\alpha}{5} + \frac{F^{(2)}}{135} + \frac{2F^{(4)}}{1089} - \frac{175F^{(6)}}{42471} + \frac{nT^{(2)}}{70\sqrt{2}} - \frac{T^{(2)}}{35\sqrt{2}} \tag{107}$$

$$\alpha_\perp = \frac{4\alpha}{5} \tag{108}$$

$$\beta_\perp = -4\alpha - \frac{\beta}{6} \tag{109}$$

$$\gamma_\perp = \frac{8\alpha}{5} + \frac{\beta}{15} + \frac{2\gamma}{25} \tag{110}$$

$$T^{(2)}_\perp = T^{(2)} \tag{111}$$

(In general, if a new operator basis $\hat{O}'$ is defined by a linear transformation $\mathcal{M}$ of the original basis $\hat{O}$ such that $\hat{O}' = \mathcal{M}\hat{O}$, then for a given linear combination of the original operator basis, $\mathcal{H} = \vec{\eta}\cdot\hat{O}$, a new linear combination $\vec{\eta}'$ of the new operator basis $\hat{O}'$, can be defined such that $\vec{\eta}\cdot\hat{O} = \vec{\eta}'\cdot\hat{O}'$ by taking $\vec{\eta}' = \left(\mathcal{M}^{-1}\right)^T\vec{\eta}$.)

Using these coefficients and their accompanying operators, the semi-empirical Hamiltonian is now

$$\begin{aligned}\hat{\mathcal{H}}^\perp_{\text{eff}} = \hat{\mathcal{H}}_0 &+ \sum_{k=0,2,4,6} E^{(k)}_\perp \hat{e}^\perp_k + \zeta\sum_{i=1}^{N}\left(\hat{s}_i\cdot\hat{l}_i\right)\\ &+ \alpha_\perp\hat{\alpha}^\perp + \beta_\perp\hat{\beta}^\perp + \gamma_\perp\hat{\gamma}^\perp\\ &+ T^{(2)}_\perp\hat{t}^\perp_2 + \sum_{\substack{k=2,3,4,6,7,8,\\ 11,12,14,15,\\ 16,17,18,19}} T^{(k)}\hat{t}_k\\ &+ \sum_{k=2,4,6} P^{(k)}\hat{p}_k + \sum_{k=0,2,4} \mathcal{M}^{(k)}\hat{m}_k\\ &+ \sum_{i=1}^{N}\sum_{k=2,4,6}\sum_{q=-k}^{k} \mathcal{B}^{(k)}_q \mathcal{C}^{(k)}_q(i).\end{aligned} \tag{112}$$

Some operators remain unchanged in this parametrization, specifically $\hat{C}^k_q$, $\hat{t}_k$ ($k \neq 2$), and the spin-orbit term. However some operators are still non-orthogonal. Operators from different *families* are mutually orthogonal, and all pairs within each family are orthogonal as well. Nevertheless, any two operators from either $\hat{m}_k$ or $\hat{p}_k$ are non-orthogonal. This residual non-orthogonality in the Hamiltonian can be resolved using the $\hat{z}_i$ operators, originally introduced by Judd in his discussion of intra-atomic magnetic interactions [JCC68]. This parametrization, which leaves $\hat{m}_k$ or $\hat{p}_k$ as they are, is therefore not entirely orthogonal, and we refer to it as the *mostly*-orthogonal Hamiltonian.

In **qlanth** the symbolic array that represents the *mostly*-orthogonal Hamiltonian can be obtained with the adequate setting of the option `OperatorBasis` in `EffectiveHamiltonian`. In that option, the standard operator basis (i.e. the one use by Carnall *et al.* [Car+89]) is termed the *legacy* basis.

```
1 EffectiveHamiltonian::usage = "EffectiveHamiltonian[numE] returns the
     Hamiltonian matrix for the f^numE configuration. The matrix is
    returned as a SparseArray.
2 The function admits an optional parameter ''FilenameAppendix'' which
    can be used to control which variant of the JJBlocks is used to
    assemble the matrix.
```

[18] Two operators $\hat{\mathcal{O}}_1$ and $\hat{\mathcal{O}}_2$ are orthogonal if $\mathrm{tr}(\hat{\mathcal{O}}^\dagger_1\hat{\mathcal{O}}_2) = 0$.

```
It also admits an optional parameter ''IncludeZeeman'' which can be used to include the Zeeman interaction. The default is False.
The option ''Set t2Switch'' can be used to toggle on or off setting the t2 selector automatically or not, the default is True, which replaces the parameter according to numE.
The option ''ReturnInBlocks'' can be used to return the matrix in block or flattened form. The default is to return it in flattened form.";
Options[EffectiveHamiltonian] = {
      "FilenameAppendix" -> "",
      "IncludeZeeman"    -> False,
      "Set t2Switch"     -> True,
      "ReturnInBlocks"   -> False,
      "OperatorBasis"    -> "Legacy"};
EffectiveHamiltonian[nf_, OptionsPattern[]] := Module[
  {numE, ii, jj, howManyJs, Js, blockHam, opBasis},
  (
    (*######################################*)
    ImportFun = ImportMZip;
    opBasis = OptionValue["OperatorBasis"];
    If[Not[MemberQ[{"Legacy", "MostlyOrthogonal", "Orthogonal"}, opBasis]],
      Echo["Operator basis " <> opBasis <> " not recognized, using ''Legacy'' basis."];
      opBasis = "Legacy";
    ];
    If[opBasis == "Orthogonal",
      Echo["Operator basis ''Orthogonal'' not implemented yet, aborting ..."];
      Return[Null];
    ];
    (*######################################*)
    If[opBasis == "MostlyOrthogonal",
      (
        blockHam = EffectiveHamiltonian[nf,
        "OperatorBasis" -> "Legacy",
        "FilenameAppendix" -> OptionValue["FilenameAppendix"],
        "IncludeZeeman" -> OptionValue["IncludeZeeman"],
        "Set t2Switch" -> OptionValue["Set t2Switch"],
        "ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
        paramChanger = Which[
          nf < 7,
          <|
            F0 -> 1/91 (54 E1p+91 E0p+78 γp),
            F2 -> (15/392 *
              (
                140   E1p  +
                20020 E2p +
                1540  E3p  +
                770 αp -
                70 γp  +
                22 Sqrt[2] T2p -
                11 Sqrt[2] nf T2p t2Switch -
                11 Sqrt[2] (14-nf) T2p (1 - t2Switch)
              )
            ),
            F4 -> (99/490 *
              (
                70   E1p -
                9100 E2p +
                280  E3p +
                140 αp -
                35 γp  +
                4 Sqrt[2] T2p -
                2 Sqrt[2] nf T2p t2Switch -
                2 Sqrt[2] (14-nf) T2p (1-t2Switch)
              )
              ),
            F6 -> (5577/7000 *
              (
                20  E1p +
                700 E2p -
                140 E3p -
                70 αp -
                10 γp -
                2 Sqrt[2] T2p +
```

```
71                Sqrt[2] nf T2p t2Switch +
72                Sqrt[2] (14-nf) T2p (1-t2Switch)
73              )
74              ),
75            ζ  -> ζ,
76            α -> (5 αp)/4,
77            β  -> -6  (5 αp + βp),
78            γ -> 5/2 (2 βp + 5 γp),
79            T2 -> 0
80          |>,
81        nf >= 7,
82          <|
83            F0 -> 1/91 (54 E1p+91 E0p+78 γp),
84            F2 -> (15/392 *
85              (
86                140   E1p  +
87                20020 E2p +
88                1540  E3p  +
89                770 αp -
90                70 γp  +
91                22 Sqrt[2] T2p -
92                11 Sqrt[2] nf T2p
93              )
94            ),
95            F4 -> (99/490 *
96              (
97                70   E1p -
98                9100 E2p +
99                280  E3p +
100               140 αp -
101               35 γp  +
102               4 Sqrt[2] T2p -
103               2 Sqrt[2] nf T2p
104             )
105             ),
106           F6 -> (5577/7000 *
107             (
108               20  E1p +
109               700 E2p -
110               140 E3p -
111               70 αp -
112               10 γp -
113               2 Sqrt[2] T2p +
114               Sqrt[2] nf T2p
115             )
116             ),
117           ζ  -> ζ,
118           α -> (5 αp)/4,
119           β  -> -6  (5 αp + βp),
120           γ -> 5/2 (2 βp + 5 γp),
121           T2 -> 0
122         |>
123       ];
124       blockHamM0 = Which[
125         OptionValue["ReturnInBlocks"] == False,
126         ReplaceInSparseArray[blockHam, paramChanger],
127         OptionValue["ReturnInBlocks"] == True,
128         Map[ReplaceInSparseArray[#, paramChanger]&, blockHam, {2}]
129       ];
130       Return[blockHamM0];
131     )
132   ];
133   (*########################################*)
134
135   (*hole-particle equivalence enforcement*)
136   numE = nf;
137   allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2, T2p
      ,
138     T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
139     α, β, γ, B02, B04, B06, B12, B14, B16,
140     B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16, S22
      ,
141     S24, S26, S34, S36, S44, S46, S56, S66, T11p, T12, T14, T15,
      T16,
142     T17, T18, T19, Bx, By, Bz};
143   params0 = AssociationThread[allVars, allVars];
```

```
    If[nf > 7,
      (
        numE = 14 - nf;
        params = HoleElectronConjugation[params0];
        If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
      ),
      params = params0;
      If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
    ];
    (* Load symbolic expressions for LS,J,J' energy sub-matrices. *)
    emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
    OptionValue["FilenameAppendix"]];
    JJBlockMatrixTable = ImportFun[emFname];
    (*Patch together the entire matrix representation using J,J'
    blocks.*)
    PrintTemporary["Patching JJ blocks ..."];
    Js         = AllowedJ[numE];
    howManyJs = Length[Js];
    blockHam  = ConstantArray[0, {howManyJs, howManyJs}];
    Do[
      blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[jj
    ]]}];,
    {ii, 1, howManyJs},
    {jj, 1, howManyJs}
    ];
    (* Once the block form is created flatten it *)
    If[Not[OptionValue["ReturnInBlocks"]],
      (blockHam = ArrayFlatten[blockHam];
      blockHam  = ReplaceInSparseArray[blockHam, params];
      ),
      (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
    ,{2}];)
    ];

    If[OptionValue["IncludeZeeman"],
      (
        PrintTemporary["Including Zeeman terms ..."];
        {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
    ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
        blockHam += - teslaToKayser * (Bx * magx + By * magy + Bz *
    magz);
      )
    ];
    Return[blockHam];
  )
];
```

4 Coordinate system

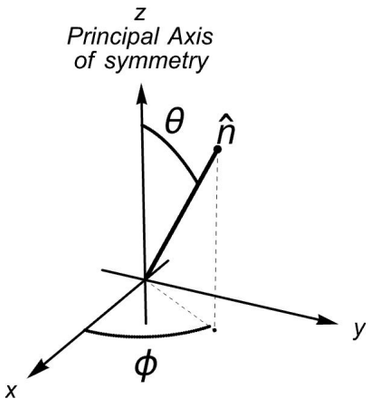
Before adding interactions lacking spherical symmetry, the orientation of the coordinate system is irrelevant. At the point when the crystal field is added, this orientation becomes relevant in the sense that only certain orientations of the coordinate system yield an expression for the crystal field potential in its simplest form[19]. To accomplish this the z-axis needs to be taken as one of the principal axis of symmetry[20]. To complete the orientation of the coordinate system, the x and y axes are chosen as symmetry axes perpendicular to the z-axis. Furthermore, certain choices for the orientation of the coordinate system also allow one to make certain crystal field parameters real, or to fix their sign.

5 Spectroscopic measurements and uncertainty

We may categorize the uncertainty in the parameters fitted to experimental data in three categories: experimental, model, and others.

Before listing the sources that contribute to experimental error, let's briefly recount the types of experiments that are used in order to determine level energies and state labels. The first type is absorption spectroscopy, in which a crystal, adequately doped, is illuminated with a broad spectrum light source, and the wavelength dependent absorption

[19] Of course, the crystal field potential can be expressed in any rotated coordinate system, but in these the potential would include additional $C_q^{(k)}$ with linear combinations of the $B_q^{(k)}$ [20] A principal axis is a symmetry axis having the most rotational symmetry in the relevant symmetry group. For example, in cubic groups, the principal axis is the 111 diagonal.



by the crystal thus determined. The crystals absorb radiation depending on the availability of a transition energy between the thermally populated low-lying states and excited levels. This data therefore provides transition energies between the ground multiplet and excited states. Furthermore, from this data one can also estimate the probability that a photon of a given wavelength is absorbed by the ions in the crystal, and from this one estimates the oscillator strength of a given transition.

In order to inform to what two multiplets the transitions belong to, here one may already count how many lines arrange themselves in groups. Alas, this type of absorption spectroscopy is lacking in that it only provides information about transitions between the ground and excited states, and may even elide such transitions that are too weak to be observed. In view of this, some variety of emission spectroscopy becomes relevant. In these, the ions inside of the crystal are excited (thermally, electrically, or radiatively) and the light produced by the relaxing transitions are then registered. This has the benefit that one has now populated other states than the ground state (perhaps with aid of non-radiative transitions inside of the crystal or energy transfer) so that now one can also have information about transitions that depart from a state different than ground. From this type of spectroscopy, given a transition, one may also determine its transition rate, given the availability of time-resolved emission; given this one may then give an upper bound on the spontaneous rate of identified transitions.

In these analyses a few things may not go according to plan:

1. **Several non-equivalent symmetry sites**. Ions may not be located in sites with the same crystal symmetry. As such it will be problematic to interpret their crystal splittings based on the assumption of a single symmetry.

2. **Non-homogeneous crystal field**. Even if they are located in sites with the same point symmetry, it may also be that the crystal field they experience has variations across the bulk of the crystal. As such, the observations would then rather be about an ensemble of crystal fields, instead of a single one.

3. **Crystal impurities**. The doped crystals may contain impurities that will lead to the false identification of transitions to the ion of study. This may be disambiguated from pooling together several experiments.

4. **Non-radiative transitions**, mediated by the crystal, will lead to shifts in transition energies, both in emission, and in absorption. This yields a confounding factor for the *radiative* transition energies that are in principle required to be valid inputs to the model Hamiltonian. Comparison of emission and absorption lines is key to determine the relevance of this.

5. **Spectrometer resolution**. The spectrometers used have a finite resolution. In the setups typically used for this, the nominal resolution might be of the order of 0.1 nm.

6. **Crystal transparency**. Observation of transitions within the ions requires that the crystal be mostly transparent at the relevant wavelengths.

In the works of Carnall and others, the nominal uncertainty in the state energies is of $1\mathcal{K}$, this being the precision to which the used experimental energies are quoted.

With regards to model uncertainties, the following factors may be considered to contribute to it:

1. **Intra-configuration transitions**. When energy levels reach a certain threshold, observations may no longer be intra-configuration transitions, but rather inter-configuration transitions. These transitions should not be included, so care must be taken to exclude them from the analysis.

2. **Unaccounted configuration interaction**. The model makes an attempt at describing configuration interaction effects, but this is only carried to second order in the types of considered interactions, and not all interactions are considered.

Finally, in the "others" category we have the two following:

1. **Numerical precision.** No longer relevant with modern computers, however, at the time at which some of these calculations were done, numerical precision might account for some of the discrepancies one finds when comparing current calculations to old ones.

2. **Errors in tables with reduced matrix elements**. The Crosswhite group at Argonne National Lab produced a set of tables with the reduced matrix elements of operators. However, at some point, these tables became slightly corrupted, and subsequent codes that used them carried those errors with them. In `qlanth` this problem is avoided since all reduced matrix elements are calculated from scratch.

When the model parameters are fitted to experimental data and their uncertainties are being estimated, `qlanth` offers two approaches. In the first approach a given constant uncertainty in the energy levels is assumed, this in turns determines the relevant contour of χ^2, and from this the uncertainties in the model parameters are calculated.

In an alternative approach, the uncertainties are determined *a posteriori*. The model parameters are fit to minimize the square differences between calculated energies and the experimental ones. Then, a single experimental uncertainty is assumed in all the energy levels, and taken equal to the minimum root mean square error, as taken over the available degrees of freedom. This uncertainty σ together with a chosen confidence interval p is then used to determine the contours of χ^2, which in turn determine the corresponding confidence interval in the model parameters. In a sense, the model is assumed to be valid, and the resulting uncertainties in the model parameters are adjusted to allow for this possibility.

In the examples included in this code the uncertainty in the experimental data was assumed to be constant and equal to $1\,\mathrm{cm}^{-1}$. And when the data for magnetic dipole transitions was calculated, an uncertainty equal to the σ of the related parametric fit was assumed.

6 Transitions

`qlanth` can also compute magnetic dipole transition rates within states and levels, as well as forced electric dipole transition rates between levels.

6.1 Level description

6.1.1 Forced electric dipole transitions

Any two eigenfunctions that are approximated within the limits of a single configuration cannot help but have the same parity as they are spanned by basis vectors with definite and shared parity. Analysis of the amplitudes for different transition operators can then inform as to what transitions are forbidden, which are those in which the product of the parity of the two participating wavefunctions and that of the transition operator results in odd parity. As such, within the single configuration approximation, since the product of the two participating wavefunctions is always even, then any transition described by an operator of odd parity is forbidden. This is the content of Laporte's parity selection rule. Since the parity of the magnetic dipole operator is even [21], then this operator allows for intra-configuration transitions, and since the parity of the electric dipole operator is odd, then these types of intra-configuration transitions are forbidden.

However, much as configuration interaction is an essential component in the description of the electronic structure, it has a bearing on the energy spectrum and the intra-configuration wavefunctions themselves. Configuration interaction may also be used to

[21] The parity of the electric quadrupole operator is also even, but we haven't included it in `qlanth`

bring back into the analysis the fact that the *actual* wavefunctions will also have at least a small part of them in other configurations, even if most of them may be within the ground configuration. It is therefore the case that the *actual* parity of the wavefunctions is mixed, and therefore intra-configuration [22] electric dipole transitions are actually allowed. These electric dipole transitions are called *forced* electric dipole transitions.

Judd [Jud62] and Ofelt [Ofe62] came separately to similar versions of this analysis, and showed after a series of approximations that the forced electric dipole transitions could be described by the intra-configuration matrix elements of the multi-electron unit operators $\hat{U}^{(k)}$ (for k=2,4,6) together with a set of three accompanying coefficients $\{\Omega_{(2)}, \Omega_{(4)}, \Omega_{(6)}\}$. These coefficients have a definite form related to the overlap between the mixed parity parts of the corrected wavefunctions, but they can also be considered as additional phenomenological parameters.

Judd-Ofelt theory is based on the level description, and its mathematical expression is the following. Given two intermediate coupling levels $|\alpha SLJ\rangle$ and $|\alpha' S'L'J'\rangle$, the oscillator strength between them is approximated as [Jud62]

$$\mathrm{f}_{\text{f-ED}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S'L'J'\rangle\right) = \mathcal{R}\frac{8\pi^2 m_e}{3h}\frac{\nu}{2J+1}\frac{\chi}{n}\sum_{k=2,4,6}\Omega_{(k)}\left|\langle \underline{\mathrm{f}}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\mathrm{f}}^n \alpha' S'L'J'\rangle\right|^2, \tag{113}$$

where ν is the frequency of the transition, χ the local field correction, n the refractive index of the crystal host, and $\mathcal{R} = 1$ in the case of absorption and $\mathcal{R} = n^2$ in the case of emission.

The local field correction χ accounts for the difference between the macroscopic and microscopic electric fields, in the case of ions embedded for crystals the most common choice is

$$\chi = \frac{n^2+2}{3} \tag{114}$$

and for other environments (or emitters other than ions such as molecules) different alternatives are relevant (see [DR06]).

In **qlanth** Judd-Ofelt theory is implemented with help of the functions JuddOfeltUkSquared and LevelElecDipoleOscillatorStrength.

```
JuddOfeltUkSquared::usage = "JuddOfeltUkSquared[numE, params]
    calculates the matrix elements of the Uk operator in the level
    basis. These are calculated according to equation (7) in Carnall
    1965.
The function returns a list with the following elements:
  - basis : A list with the allowed {SL, J} terms in the f^numE
    configuration. Equal to BasisLSJ[numE].
  - eigenSys : A list with the eigensystem of the Hamiltonian for the
     f^n configuration.
  - levelLabels : A list with the labels of the major components of
    the level eigenstates.
  - LevelUkSquared : An association with the squared matrix elements
    of the Uk operators in the level eigenbasis. The keys being {2, 4,
     6} corresponding to the rank of the Uk operator. The basis in
    which the matrix elements are given is the one corresponding to
    the level eigenstates given in eigenSys and whose major SLJ
    components are given in levelLabels. The matrix is symmetric and
    given as a SymmetrizedArray.
  The function admits the following options:
  ''PrintFun'' : A function that will be used to print the progress
    of the calculations. The default is PrintTemporary.";
Options[JuddOfeltUkSquared] = {"PrintFun" -> PrintTemporary};
JuddOfeltUkSquared[numE_, params_, OptionsPattern[]] := Module[
  {eigenChanger, numEH, basis, eigenSys,
  Js, Ukmat, LevelUkSquared, kRank,
  S, L, Sp, Lp, J, Jp, phase,
  braTerm, ketTerm, levelLabels,
  eigenVecs, majorComponentIndices},
  (
    If[Not[ValueQ[ReducedUkTable]],
      LoadUk[]
    ];
    numEH = Min[numE, 14-numE];
    PrintFun = OptionValue["PrintFun"];
    PrintFun["> Calculating the levels for the given parameters ..."
   ];
```

[22] Calling these *intra*-configuration transitions is somewhat of a misnomer since their nature is tied to the fact that the single-configuration description is wanting.

```
    {basis, majorComponents, eigenSys} = LevelSolver[numE, params];
    (* The change of basis matrix to the eigenstate basis *)
    eigenChanger = Transpose[Last /@ eigenSys];
    PrintFun["Calculating the matrix elements of Uk in the physical coupling basis ..."];
    LevelUkSquared = <||>;
    Do[(
      Ukmat = Table[(
          {S, L}   = FindSL[braTerm[[1]]];
          J        = braTerm[[2]];
          Jp       = ketTerm[[2]];
          {Sp, Lp} = FindSL[ketTerm[[1]]];
          phase    = Phaser[S + Lp + J + kRank];
          Simplify @ (
            phase *
            Sqrt[TPO[J]*TPO[Jp]] *
            SixJay[{J, Jp, kRank}, {Lp, L, S}] *
            ReducedUkTable[{numEH, 3, braTerm[[1]], ketTerm[[1]], kRank}]
            )
          ),
        {braTerm, basis},
        {ketTerm, basis}
      ];
      Ukmat = (Transpose[eigenChanger] . Ukmat . eigenChanger)^2;
      Ukmat = Chop@Ukmat;
      LevelUkSquared[kRank] = SymmetrizedArray[Ukmat, Dimensions[eigenChanger], Symmetric[{1, 2}]];
      ),
      {kRank, {2, 4, 6}}
    ];
    LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[InputForm[#[[2]]]]) & /@ basis;
    eigenVecs = Last /@ eigenSys;
    majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ eigenVecs;
    levelLabels = LSJmultiplets[[majorComponentIndices]];
    Return[{basis, eigenSys, levelLabels, LevelUkSquared}];
  )
];
```

```
LevelElecDipoleOscillatorStrength::usage = "
    LevelElecDipoleOscillatorStrength[numE, levelParams, juddOfeltParams] uses Judd-Ofelt theory to estimate the forced electric dipole oscillator strengths ions whose level description is determined by levelParams.
The third parameter juddOfeltParams is an association with keys equal to the three Judd-Ofelt intensity parameters {\[CapitalOmega]2, \[CapitalOmega]4, \[CapitalOmega]6} and corresponding values in cm^2.
The local field correction implemented here corresponds to the one given by the virtual cavity model of Lorentz.
The function returns a list with the following elements:
  - basis : A list with the allowed {SL, J} terms in the f^numE configuration. Equal to BasisLSJ[numE].
  - eigenSys : A list with the eigensystem of the Hamiltonian for the f^n configuration in the level description.
  - levelLabels : A list with the labels of the major components of the calculated levels.
  - oStrengthArray : A square array whose elements represent the oscillator strengths between levels such that the element oStrengthArray[[i,j]] is the oscillator strength between the levels |Subscript[\[Psi], i]> and |Subscript[\[Psi], j]>. In this array, the elements below the diagonal represent emission oscillator strengths, and elements above the diagonal represent absorption oscillator strengths.
The function admits the following three options:
  ''PrintFun'' : A function that will be used to print the progress of the calculations. The default is PrintTemporary.
  ''RefractiveIndex'' : The refractive index of the medium where the transitions are taking place. This may be a number or a function. If a number then the oscillator strengths are calculated for assuming a wavelength-independent refractive index. If a function then the refractive indices are calculated accordingly to the wavelength of each transition (the function must admit a single argument equal to the wavelength in nm). The default is 1.
  ''LocalFieldCorrection'' : The local field correction to be used.
```

```
    The default is ''VirtualCavity''. The options are: ''VirtualCavity
    '' and ''EmptyCavity''.
The equation implemented here is the one given in eqn. 29 from the
    review article of Hehlen (2013). See that same article for a
    discussion on the local field correction.
";
Options[LevelElecDipoleOscillatorStrength]={
  "PrintFun"        -> PrintTemporary,
  "RefractiveIndex" -> 1,
  "LocalFieldCorrection" -> "VirtualCavity"
};
LevelElecDipoleOscillatorStrength[numE_, levelParams_Association,
    juddOfeltParams_Association, OptionsPattern[]] := Module[
  {PrintFun, basis, eigenSys, levelLabels,
  LevelUkSquared, eigenEnergies, energyDiffs,
  oStrengthArray, nRef, \[Chi], nRefs,
  \[Chi]OverN, groundLevel, const,
  transitionFrequencies, wavelengthsInNM,
  fieldCorrectionType},
  (
    PrintFun = OptionValue["PrintFun"];
    nRef     = OptionValue["RefractiveIndex"];
    PrintFun["Calculating the Uk^2 matrix elements for the given
    parameters ..."];
    {basis, eigenSys, levelLabels, LevelUkSquared} =
    JuddOfeltUkSquared[numE, levelParams, "PrintFun"->PrintFun];
    eigenEnergies = First/@eigenSys;
    (* converted to cm^-2 *)
    const         = (8\[Pi]^2)/3 me/hPlanck * 10^(-4);
    energyDiffs   = Transpose@Outer[Subtract,eigenEnergies,
    eigenEnergies];
    (* since energies are assumed in Kayser, speed of light needs to
    be in cm/s, so that the frequencies are in 1/s *)
    transitionFrequencies = energyDiffs*cLight*100;
    (* grab the J for each level *)
    levelJs        = #[[2]] & /@ eigenSys;
    oStrengthArray = (
      juddOfeltParams[\[CapitalOmega]2]*LevelUkSquared[2]+
      juddOfeltParams[\[CapitalOmega]4]*LevelUkSquared[4]+
      juddOfeltParams[\[CapitalOmega]6]*LevelUkSquared[6]
    );
    oStrengthArray = Abs@(const * transitionFrequencies *
    oStrengthArray);
    (* it is necessary to divide each oscillator strength by the
    degeneracy of the initial level *)
    oStrengthArray = MapIndexed[1/(2 levelJs[[#2[[1]]]]+1) #1 &,
    oStrengthArray,{2}];
    (* including the effects of the refractive index *)
    fieldCorrectionType = OptionValue["LocalFieldCorrection"];
    Which[
      nRef === 1,
        True,
      NumberQ[nRef],
      (
        \[Chi] = Which[
          fieldCorrectionType == "VirtualCavity",
          (
            ( (nRef^2 + 2) / 3 )^2
          ),
          fieldCorrectionType == "EmptyCavity",
          (
            ( 3 * nRef^2 / ( 2 * nRef^2 + 1 ) )^2
          )
        ];
        \[Chi]OverN    = \[Chi] / nRef;
        oStrengthArray = \[Chi]OverN * oStrengthArray;
        (* the refractive index participates differently in
    absorption and in emission *)
        aFunction      = If[#2[[1]]>#2[[2]], #1 * nRef^2, #1]&;
        oStrengthArray = MapIndexed[aFunction, oStrengthArray, {2}];
      ),
      True,
      (
        wavelengthsInNM = Abs[1 / energyDiffs] * 10^7;
        nRefs           = Map[nRef, wavelengthsInNM];
        Echo["Calculating the oscillator strengths for the given
```

```
    refractive index ..."];
76        \[Chi] = Which[
77          fieldCorrectionType == "VirtualCavity",
78          (
79            ( (nRefs^2 + 2) / 3 )^2
80          ),
81          fieldCorrectionType == "EmptyCavity",
82          (
83            ( 3 * nRefs^2 / ( 2*nRefs^2 + 1 ) )^2
84          )
85        ];
86        \[Chi]OverN     = \[Chi] / nRefs;
87        oStrengthArray  = \[Chi]OverN * oStrengthArray
88      )
89    ];
90    Return[{basis, eigenSys, levelLabels, oStrengthArray}];
91  )
92 ];
```

6.1.2 Magnetic dipole transitions

In Hartree atomic units, the magnetic dipole line strength between levels $|\alpha LSJ\rangle$ and $|\alpha' S' L' J'\rangle$ is given by

$$\hat{S}_{\mathrm{MD}}(|\alpha LSJ\rangle, |\alpha' S' L' J'\rangle) = \left| \langle \alpha LSJ \| \frac{1}{2}(\hat{L} + g\hat{S}) \| \alpha' S' L' J' \rangle \right|^2 \tag{115}$$

In **qlanth** the line strength can be calculated using the function LevelMagDipoleLineStrength.

```
1  LevelMagDipoleLineStrength::usage = "LevelMagDipoleLineStrength[
       eigenSys, numE] calculates the magnetic dipole line strengths for
       an ion whose level description is determined by levelParams. The
       function returns a square array whose elements represent the
       magnetic dipole line strengths between the levels given in
       eigenSys such that the element magDipoleLineStrength[[i,j]] is the
        line strength between the levels |Subscript[\[Psi], i]> and |
       Subscript[\[Psi], j]>. Eigensys must be such that it consists of a
        lists of lists where in each list the last element corresponds to
        the eigenvector of a level (given as a row) in the standard basis
        for levels of the f^numE configuration.
2  The function admits the following options:
3    ''Units'' : The units in which the line strengths are given. The
       default is ''SI''. The options are ''SI'' and ''Hartree''. If ''SI
       '' then the unit of the line strength is (A m^2)^2 = (J/T)^2. If
       ''Hartree'' then the line strength is given in units of 2 \[Mu]B."
       ;
4  Options[LevelMagDipoleLineStrength] = {
5    "Units"->"SI"
6    };
7  LevelMagDipoleLineStrength[theEigensys_List, numE0_Integer,
       OptionsPattern[]] := Module[
8    {numE, levelMagOp, allEigenvecs, magDipoleLineStrength, units},
9    (
10     numE          = Min[14-numE0,numE0];
11     levelMagOp    = LevelMagDipoleMatrixAssembly[numE];
12     allEigenvecs  = Transpose[Last/@theEigensys];
13     units         = OptionValue["Units"];
14     magDipoleLineStrength          = Transpose[allEigenvecs].
       levelMagOp.allEigenvecs;
15     magDipoleLineStrength          = Abs[magDipoleLineStrength]^2;
16     Which[
17       units=="SI",
18         Return[4 \[Mu]B^2 * magDipoleLineStrength],
19       units=="Hartree",
20         Return[magDipoleLineStrength]
21     ];
22   )
23 ];
```

In atomic units, the magnetic dipole oscillator strength for a transition between level $|\alpha LSJ\rangle$ and an excited level $|\alpha' S' L' J'\rangle$ is given by [Rud07]

$$\mathrm{f}_{\mathrm{MD}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S' L' J'\rangle\right) = \frac{2n}{3} \frac{\mathcal{E}(|\alpha' S' L' J'\rangle) - \mathcal{E}(|\alpha LSJ\rangle)}{2J+1} \alpha^2 \hat{S}_{\mathrm{MD}}\left(|\alpha LSJ\rangle, |\alpha' S' L' J'\rangle\right) \tag{116}$$

where $\mathcal{E}(|\alpha LSJ\rangle)$ is the energy of level $|\alpha LSJ\rangle$, n is the refractive index of the medium, and α is the fine structure constant. In obtaining this expression one considers the transition from one state of the initial level into another single state of the final level. Furthermore, here it is assumed that all the states of the initial level are equally populated.

In **qlanth** the function LevelMagDipoleOscillatorStrength can be used to calculate these.

```
LevelMagDipoleOscillatorStrength::usage = "
    LevelMagDipoleOscillatorStrength[eigenSys, numE] calculates the
    magnetic dipole oscillator strengths for an ion whose levels are
    described by eigenSys in configuration f^numE. The refractive
    index of the medium is relevant, but here it is assumed to be 1,
    this can be changed through the option ''RefractiveIndex''.
    eigenSys must consist of a lists of lists with three elements: the
     first element being the energy of the level, the second element
    being the J of the level, and the third element being the
    eigenvector of the level.
  The function returns a list with the following elements:
    - basis : A list with the allowed {SL, J} terms in the f^numE
    configuration. Equal to BasisLSJ[numE].
    - eigenSys : A list with the eigensystem of the Hamiltonian for
    the f^n configuration in the level description.
    - magDipoleOstrength : A square array whose elements represent
    the magnetic dipole oscillator strengths between the levels given
    in eigenSys such that the element magDipoleOstrength[[i,j]] is the
     oscillator strength between the levels |Subscript[\[Psi], i]> and
     |Subscript[\[Psi], j]>. In this array the elements below the
    diagonal represent emission oscillator strengths, and elements
    above the diagonal represent absorption oscillator strengths. The
    emission oscillator strengths are negative. The oscillator
    strength is a dimensionless quantity.
  The function admits the following option:
    ''RefractiveIndex'' : The refractive index of the medium where
    the transitions are taking place. This may be a number or a
    function. If a number then the oscillator strengths are calculated
     assuming a wavelength-independent refractive index as given. If a
     function then the refractive indices are calculated accordingly
    to the vaccum wavelength of each transition (the function must
    admit a single argument equal to the wavelength in nm). The
    default is 1.
  For reference see equation (27.8) in Rudzikas (2007). The
    expression for the line strenght is the simplest when using atomic
     units, (27.8) is missing a factor of \[Alpha]^2.";
Options[LevelMagDipoleOscillatorStrength]={
  "RefractiveIndex" -> 1
};
LevelMagDipoleOscillatorStrength[eigenSys_, numE_, OptionsPattern[]]
    := Module[
  {eigenEnergies, eigenVecs, levelJs,
  energyDiffs, magDipoleOstrength, nRef,
  wavelengthsInNM, nRefs, degenDivisor},
  (
    basis          = BasisLSJ[numE];
    eigenEnergies  = First/@eigenSys;
    nRef           = OptionValue["RefractiveIndex"];
    eigenVecs      = Last/@eigenSys;
    levelJs        = #[[2]]&/@eigenSys;
    energyDiffs    = -Outer[Subtract,eigenEnergies,eigenEnergies];
    energyDiffs  *= kayserToHartree;
    magDipoleOstrength = LevelMagDipoleLineStrength[eigenSys, numE, "
    Units"->"Hartree"];
    magDipoleOstrength = 2/3 * \[Alpha]Fine^2 * energyDiffs *
    magDipoleOstrength;
    degenDivisor       = #1 / ( 2 * levelJs[[#2[[1]]]] + 1 ) &;
    magDipoleOstrength = MapIndexed[degenDivisor, magDipoleOstrength,
     {2}];
    Which[nRef===1,
      True,
      NumberQ[nRef],
        (
          magDipoleOstrength = nRef * magDipoleOstrength;
        ),
      True,
        (
          wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
    10^7;
```

```
            nRefs                = Map[nRef, wavelengthsInNM];
            magDipoleOstrength   = nRefs * magDipoleOstrength;
          )
      ];
      Return[{basis, eigenSys, magDipoleOstrength}];
    )
];
```

A final quantity of interest is the spontaneous magnetic dipole decay rate from one level to a lower lying one. In atomic units this rate is determined by

$$\Gamma_{\mathrm{MD}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S' L' J'\rangle\right) = \frac{4n^3}{3}\frac{\left(\mathcal{E}(|\alpha LSJ\rangle) - \mathcal{E}(|\alpha' S' L' J'\rangle)\right)^3}{2J+1}\alpha^5 \hat{\mathcal{S}}_{\mathrm{MD}}\left(|\alpha LSJ\rangle, |\alpha' S' L' J'\rangle\right). \tag{117}$$

In **qlanth** the spontaneous decay rates may be calculated through the function LevelMagDipoleSpotaneousDecayRates.

```
LevelMagDipoleSpotaneousDecayRates::usage = "
    LevelMagDipoleSpotaneousDecayRates[eigenSys, numE] calculates the
    spontaneous emission rates for the magnetic dipole transitions
    between the levels given in eigenSys. The function returns a
    square array whose elements represent the spontaneous emission
    rates between the levels given in eigenSys such that the element
    [[i,j]] of the returned array is the rate of spontaneous emission
    from the level |Subscript[\[Psi], i]> to the level |Subscript[\[
    Psi], j]>. In this array the elements below the diagonal represent
     emission rates, and elements above the diagonal are identically
    zero.
The function admits two optional arguments:
  + \"Units\" : The units in which the rates are given. The default
    is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI\" then
    the rates are given in s^-1. If \"Hartree\" then the rates are
    given in the atomic unit of frequency.
  + \"RefractiveIndex\" : The refractive index of the medium where
    the transitions are taking place. This may be a number or a
    function. If a number then the rates are calculated assuming a
    wavelength-independent refractive index as given. If a function
    then the refractive indices are calculated accordingly to the
    vaccum wavelength of each transition (the function must admit a
    single argument equal to the wavelength in nm). The default is 1."
    ;
Options[LevelMagDipoleSpotaneousDecayRates] = {
  "Units" -> "SI",
  "RefractiveIndex" -> 1};
LevelMagDipoleSpotaneousDecayRates[eigenSys_List, numE_Integer,
    OptionsPattern[]] := Module[
  {
    levMDlineStrength, eigenEnergies, energyDiffs, levelJs,
    spontaneousRatesInHartree, spontaneousRatesInSI, degenDivisor,
    units,
    nRef, nRefs, wavelengthsInNM
  },
  (
    nRef               = OptionValue["RefractiveIndex"];
    units              = OptionValue["Units"];
    levMDlineStrength = LowerTriangularize@LevelMagDipoleLineStrength
    [eigenSys,numE,"Units"->"Hartree"];
    levMDlineStrength = SparseArray[levMDlineStrength];
    eigenEnergies      = First /@ eigenSys;
    energyDiffs        = Outer[Subtract, eigenEnergies, eigenEnergies
    ];
    energyDiffs        = kayserToHartree * energyDiffs;
    energyDiffs        = SparseArray[LowerTriangularize[energyDiffs]];
    levelJs            = #[[2]]& /@ eigenSys;
    spontaneousRatesInHartree = 4/3 αFine^5 * energyDiffs^3 *
    levMDlineStrength;
    degenDivisor               = #1 / (2*levelJs[[#2[[1]]]] + 1)&;
    spontaneousRatesInHartree = MapIndexed[degenDivisor,
    spontaneousRatesInHartree, {2}];
    Which[nRef===1,
      True,
      NumberQ[nRef],
        (
          spontaneousRatesInHartree = nRef^3 *
    spontaneousRatesInHartree;
        ),
```

```
      True,
        (
          wavelengthsInNM     = Abs[kayserToHartree / energyDiffs] *
    10^7;
          nRefs               = Map[nRef, wavelengthsInNM];
          spontaneousRatesInHartree = nRefs^3 *
    spontaneousRatesInHartree;
        )
    ];
    If[units == "SI",
      (
        spontaneousRatesInSI = 1/hartreeTime *
    spontaneousRatesInHartree;
        Return[SparseArray@spontaneousRatesInSI];
      ),
      Return[SparseArray@spontaneousRatesInHartree];
    ];
  )
];
```

6.2 State description

6.2.1 Forced electric dipole transitions

There is an extension to Judd-Ofelt theory [MRR87b; MRR87c; MRR87a; RR83a; RR84; BCR99; NB75; Rei92; Rei87; RR83b; MR92; MR90; MJR89b; MJR89a; BSR88; Bur+94] in which one may estimate the line strength between states as they are split by the crystal field. In this approach, a rank-1 parametric dipole operator is defined as [NN00]

$$\hat{\mathcal{D}}_{\text{eff},q} = -e \sum_{\lambda=\{2,4,6\}} \sum_{t=\{\lambda-1,\lambda,\lambda+1\}} \sum_{p} \mathscr{A}_{t,p}^{(\lambda)} \langle \lambda, (p+q); 1, -q | t, p \rangle \hat{U}_{p+q}^{(\lambda)}. \tag{118}$$

Where the sum over p goes over all values allowed by the Clebsch-Gordan coefficients. Where $\hat{U}_{p+q}^{(\lambda)}$ is the symmetric unit tensor operator of rank λ and with $\mathscr{A}_{t,p}^{(\lambda)}$ a set of empirical parameters to be fitted against extensive experimental data that witness the intensity of transitions between states (see [Bur+94] for an example of Nd^{3+} in YAG). The parameters $\mathscr{A}_{t,p}^{(\lambda)}$ satisfy the general constraint

$$\left(\mathscr{A}_{t,p}^{(\lambda)}\right)^* = (-1)^{t+p+1} \mathscr{A}_{t,-p}^{(\lambda)}, \tag{119}$$

with allowed p values (which further defines the sum over p in the expression for $\hat{\mathcal{D}}_{\text{eff}}$) constrained by the point symmetry of the site occupied by the ion.

In **qlanth** this may be calculated with the function `EffectiveElectricDipole`.

From this operator the forced electric dipole line strength $\hat{\mathcal{S}}_{\text{ED}}$ is calculated as

$$\hat{\mathcal{S}}_{\text{ED}}(\psi, \psi') := \left|\langle\psi|\hat{\mathcal{D}}_{\text{eff}}|\psi'\rangle\right|^2 = \left|\langle\psi|\hat{\mathcal{D}}_{\text{eff},-1}|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mathcal{D}}_{\text{eff},0}|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mathcal{D}}_{\text{eff},1}|\psi'\rangle\right|^2. \tag{120}$$

In **qlanth** this may be calculated with the function `ElectricDipLineStrength`.

Together with the magnetic dipole line strength $\hat{\mathcal{S}}_{\text{MD}}$, the total line strength $\hat{\mathcal{S}}_{\text{TOTAL}}$ is then

$$\hat{\mathcal{S}}_{\text{TOTAL}} = \hat{\mathcal{S}}_{\text{ED}} + \frac{1}{c^2} \hat{\mathcal{S}}_{\text{MD}}. \tag{121}$$

```
EffectiveElectricDipole::usage = "EffectiveElectricDipole[numE,
    Aparams] calculates the effective dipole operator in configuration
     f^numE for the given intensity parameters Aparams (given as an
    Association). The function returns an Association with three keys
    {-1,0,1} representing the three components of the operator and
    with values equal to arrays for the corresponding matrix
    representations in the standard LSJMJ basis. The units being equal
     to those of the units assumed for Aparams.";
EffectiveElectricDipole[numE0_, Aparams0_] := Module[
  {
    Aparams = Aparams0,
    numE = numE0,
    Deff,
    keys,
    key,
    altKey,
    Aparam,
```

```
    qComponents
    },
  (
    keys = Keys[Aparams];
    Deff = <||>;
    qComponents = {-1, 0, 1};
    If[Not[ValueQ[Uks]],
      Uks = <||>
    ];
    Do[
    (
      sumTerms = Reap[
        Do[
          (
          key    = {\[Lambda], t, p};
          altKey = {\[Lambda], t, -p};
          (* the way the iterator is carried out, some CG are non-
   physical, leave them out early *)
          If[
            Or[Abs[p] > t,
            Abs[p + q] > \[Lambda]
            ],
          Continue[]
          ];
          (* enforce relation between conjugate values *)
          Aparam = Which[
            MemberQ[keys, key],
            Aparams[key],
            MemberQ[keys, altKey],
            Conjugate[Phaser[t + p + 1] * Aparams[altKey]],
            True,
            Continue[]
            ];
          (* calculate the Uk operator if it has not been calculated
   *)
          If[Not@MemberQ[Keys[Uks], \[Lambda]],
            Uks[\[Lambda]] = UkOperator[numE, \[Lambda]];
            ];

          clebscG = ClebschGordan[
            {\[Lambda], p + q},
            {1, -q},
            {t, p}];
          If[clebscG === 0, Continue[]];

          Sow[Phaser[q]*Aparam*clebscG*Uks[\[Lambda]][{\[Lambda], p +
    q}]];
          ),
          {\[Lambda], {2, 4, 6}},
          {t, {\[Lambda] - 1, \[Lambda], \[Lambda] + 1}},
          {p, -t, t, 1}
          ]
        ][[2, 1]];
      Deff[q] = Total[N@sumTerms];
      ),
    {q, qComponents}
    ];
    Return[Deff]
    )
  ];
```

```
ElectricDipLineStrength::usage = "ElectricDipLineStrength[theEigensys
    , numE, Aparams] takes the eigensystem of an ion with
    configuration f^numE together with the intensity parameters A that
     define the effective electric dipole operator.
The option ''Units'' can be set to either ''SI'' (so that the units
    of the returned array are (units of A)^2 * C^2) or to ''Hartree-
    partial'' in which case the units of the returned array are
    determined by the units of Aparams and equal to (units of Aparams)
    ^2.
The option ''States'' can be used to limit the states for which the
    line strength is calculated. The default, All, calculates the line
     strength for all states. A second option for this is to provide
    an index labelling a specific state, in which case only the line
    strengths between that state and all the others are computed.
The returned array should be interpreted in the eigenbasis of the
```

```
    Hamiltonian. As such the element Sed[[i,i]] corresponds to the
    line strength states between states |i> and |j>.
In what is returned no sum is made over degenerate states.
";
Options[ElectricDipLineStrength] = {"Units" -> "SI","States" -> All};
ElectricDipLineStrength[theEigensys_List, numE0_Integer,
    Aparams_Association, OptionsPattern[]] := Module[
  {
    numE = numE0,
    Deffcalc,
    allEigenvecs,
    Sed
  },
  (
    numE = Min[14-numE, numE];
    Deff = EffectiveElectricDipole[numE, Aparams];
    allEigenvecs = Transpose[Last /@ theEigensys];

    Which[OptionValue["States"] === All,
      (
        Deffcalc = (ConjugateTranspose[allEigenvecs] . # .
    allEigenvecs) & /@ Deff;
      ),
      IntegerQ[OptionValue["States"]],
      (
        singleState  = theEigensys[[OptionValue["States"],2]];
        Deffcalc = (ConjugateTranspose[allEigenvecs] . # .
    singleState) & /@ Deff;
      )
    ];
    Sed  = (Abs[Deffcalc[-1]^2] + Abs[Deffcalc[0]^2] + Abs[Deffcalc
    [1]^2]);
    Which[
      OptionValue["Units"] == "SI",
      Return[eCharge^2 * Sed],
      OptionValue["Units"] == "Hartree-partial",
      Return[Sed]
    ];
  )
];
```

6.2.2 Magnetic dipole transitions

qlanth can also calculate a few quantities related to magnetic dipole transitions. With $\hat{\mu} = \{\hat{\mu}_x, \hat{\mu}_y, \hat{\mu}_z\}$ the magnetic dipole operator, the line strength between two eigenstates $|\nu\rangle$ and $|\nu'\rangle$ is defined as (see for example equation 14.31 in [Cow81])

$$\hat{S}_{\mathrm{MD}}(\psi, \psi') \coloneqq \left|\langle\psi|\hat{\mu}|\psi'\rangle\right|^2 = \left|\langle\psi|\hat{\mu}_x|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_y|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_z|\psi'\rangle\right|^2 \tag{122}$$

In **qlanth** this is computed with the function `MagDipLineStrength`, which given a set of eigenvectors computes the sum above, and returns an array that contains all possible pairings of $|\psi\rangle$ and $|\psi'\rangle$ in $\hat{S}_{\mathrm{MD}}(\psi, \psi')$.

```
MagDipLineStrength::usage = "MagDipLineStrength[theEigensys, numE]
    takes the eigensystem of an ion and the number numE of f-electrons
     that correspond to it and calculates the line strength array Stot
    .
The option ''Units'' can be set to either ''SI'' (so that the units
    of the returned array are (A m^2)^2) or to ''Hartree''.
The option ''States'' can be used to limit the states for which the
    line strength is calculated. The default, All, calculates the line
     strength for all states. A second option for this is to provide
    an index labelling a specific state, in which case only the line
    strengths between that state and all the others are computed.
The returned array should be interpreted in the eigenbasis of the
    Hamiltonian. As such the element Stot[[i,i]] corresponds to the
    line strength states between states |i> and |j>.";
Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI",
    "States" -> All};
MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern[]]
     := Module[
  {numE, allEigenvecs, Sx, Sy, Sz, Stot, factor},
  (
    numE = Min[14-numE0, numE0];
```

```
    (*If not loaded then load it, *)
    If[Or[
        Not[MemberQ[Keys[magOp], numE]],
        OptionValue["Reload MagOp"]],
    (
      magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
    MagDipoleMatrixAssembly[numE];
    )
    ];
    allEigenvecs = Transpose[Last /@ theEigensys];
    Which[OptionValue["States"] === All,
      (
        {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
    allEigenvecs) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      ),
      IntegerQ[OptionValue["States"]],
      (
        singleState  = theEigensys[[OptionValue["States"],2]];
        {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
    singleState) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      )
    ];
    Which[
      OptionValue["Units"] == "SI",
        Return[4 \[Mu]B^2 * Stot],
      OptionValue["Units"] == "Hartree",
        Return[Stot],
      True,
      (
        Echo["Invalid option for ''Units''. Options are ''SI'' and ''
    Hartree''."];
        Abort[];
      )
    ];
  )
];
```

Using the line strength $\hat{\mathcal{S}}_{\mathrm{MD}}$, the transition rate A_{MD} for the spontaneous transition $|\psi_i\rangle \rightsquigarrow |\psi_f\rangle$ is then given by (from table 7.3 of [TLJ99])

$$\mathrm{A}_{\mathrm{MD}}\left(|\psi_i\rangle \rightsquigarrow |\psi_f\rangle\right) = \frac{16\pi^3\mu_0}{3h}\frac{n^3}{\lambda_{if}^3}\frac{\hat{\mathcal{S}}_{\mathrm{MD}}(\psi_i,\psi_f)}{g_i}, \tag{123}$$

where λ is the vacuum-equivalent wavelength of the transition between $|\nu\rangle$ and $|\nu'\rangle$, n the refractive index of the medium containing the ion, and g_i the degeneracy of the initial state $|\psi_i\rangle$. At the state level of description, J is no longer a good quantum number so the degeneracy $g_i = 1$.

```
MagDipoleRates::usage = "MagDipoleRates[eigenSys, numE] calculates
    the magnetic dipole transition rate array for the provided
    eigensystem. The option ''Units'' can be set to ''SI'' or to ''
    Hartree''. If the option ''Natural Radiative Lifetimes'' is set to
     true then the reciprocal of the rate is returned instead.
    eigenSys is a list of lists with two elements, in each list the
    first element is the energy and the second one the corresponding
    eigenvector.
Based on table 7.3 of Thorne 1999, using g2=1.
The energy unit assumed in eigenSys is kayser.
The returned array should be interpreted in the eigenbasis of the
    Hamiltonian. As such the element AMD[[i,i]] corresponds to the
    transition rate (or the radiative lifetime, depending on options)
    between eigenstates |i> and |j>.
By default this assumes that the refractive index is unity, this may
    be changed by setting the option ''RefractiveIndex'' to the
    desired value.
The option ''Lifetime'' can be used to return the reciprocal of the
    transition rates. The default is to return the transition rates.";
Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False, "
    RefractiveIndex"->1};
MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]] :=
    Module[
  {AMD, Stot, eigenEnergies,
  transitionWaveLengthsInMeters, nRefractive},
```

```
  (
    nRefractive   = OptionValue["RefractiveIndex"];
    numE          = Min[14-numE0, numE0];
    Stot          = MagDipLineStrength[eigenSys, numE, "Units" ->
    OptionValue["Units"]];
    eigenEnergies = Chop[First/@eigenSys];
    energyDiffs   = Outer[Subtract, eigenEnergies, eigenEnergies];
    energyDiffs   = ReplaceDiagonal[energyDiffs, Indeterminate];
    (* Energies assumed in kayser.*)
    transitionWaveLengthsInMeters = 0.01/energyDiffs;

    unitFactor    = Which[
    OptionValue["Units"]=="Hartree",
    (
      (* The bohrRadius factor in SI needed to convert the
    wavelengths which are assumed in m*)
      16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckHartree)) * bohrRadius^3
    ),
    OptionValue["Units"]=="SI",
    (
      16 \[Pi]^3 \[Mu]0/(3 hPlanck)
    ),
    True,
    (
      Echo["Invalid option for ''Units''. Options are ''SI'' and ''
    Hartree''."];
      Abort[];
    )
    ];
    AMD = unitFactor / transitionWaveLengthsInMeters^3 * Stot *
    nRefractive^3;
    Which[OptionValue["Lifetime"],
      Return[1/AMD],
      True,
      Return[AMD]
    ]
  )
];
```

A final quantity of interest is the oscillator strength for the transition between the ground state $|\psi_g\rangle$ and an excited state $|\psi_e\rangle$. The oscillator strength is a dimensionless quantity which is indicative of how strong absorption is. The oscillator strength may be defined for other initial states than the ground state, but since this is the state most likely to be populated in ordinary experimental conditions, this is the initial state that is of most frequent interest. The oscillator strength is given by [CFW65]

$$\mathrm{f}_{\mathrm{MD}}\left(|\psi_g\rangle \rightsquigarrow |\psi_e\rangle\right) = \frac{8\pi^2 m_e}{3\,hc\,e^2}\frac{n}{\lambda_{ge}}\frac{\hat{\mathcal{S}}_{\mathrm{MD}}(\psi_g,\psi_e)}{g_g} \tag{124}$$

where g_g is the degeneracy of the ground state. At the level of detail that the eigenstates are described in **qlanth** where J is no longer a good quantum number, $g_g = 1$.

In **qlanth** the function GroundMagDipoleOscillatorStrength implements the calculation of the oscillator strengths from the ground state to all the excited ones.

```
GroundMagDipoleOscillatorStrength::usage = "
    GroundMagDipoleOscillatorStrength[eigenSys, numE] calculates the
    magnetic dipole oscillator strengths between the ground state and
    the excited states as given by eigenSys.
Based on equation 8 of Carnall 1965, removing the 2J+1 factor since
    this degeneracy has been removed by the crystal field.
eigenSys is a list of lists with two elements, in each list the first
     element is the energy and the second one the corresponding
    eigenvector.
The energy unit assumed in eigenSys is Kayser.
The oscillator strengths are dimensionless.
The returned array should be interpreted in the eigenbasis of the
    Hamiltonian. As such the element fMDGS[[i]] corresponds to the
    oscillator strength between ground state and eigenstate |i>.
By default this assumes that the refractive index is unity, this may
    be changed by setting the option ''RefractiveIndex'' to the
    desired value.";
Options[GroundMagDipoleOscillatorStrength]={"RefractiveIndex"->1};
GroundMagDipoleOscillatorStrength[eigenSys_List, numE_Integer,
    OptionsPattern[]] := Module[
  {eigenEnergies, SMDGS, GSEnergy, energyDiffs,
```

```
  transitionWaveLengthsInMeters, unitFactor, nRefractive},
  (
    eigenEnergies    = First/@eigenSys;
    nRefractive      = OptionValue["RefractiveIndex"];
    SMDGS            = MagDipLineStrength[eigenSys,numE, "Units"->"SI
    ", "States"->1];
    GSEnergy         = eigenSys[[1,1]];
    energyDiffs      = eigenEnergies-GSEnergy;
    energyDiffs[[1]] = Indeterminate;
    transitionWaveLengthsInMeters = 0.01/energyDiffs;
    unitFactor       = (8\[Pi]^2 me)/(3 hPlanck eCharge^2 cLight);
    fMDGS            = unitFactor / transitionWaveLengthsInMeters *
    SMDGS * nRefractive;
    Return[fMDGS];
  )
];
```

7 Parameter constraints

When there is a scarcity of experimental data, one useful strategy to reduce the number of free parameters is to enforce some constraints between ratios of Slater integrals $F^{(k)}$, Marvin integrals $\mathcal{M}^{(k)}$, and pseudo-magnetic parameters $P^{(k)}$.

For the Slater integrals one may leave only $F^{(2)}$ as free parameter, and fix the ratios $F^{(4)}/F^{(2)}$ and $F^{(6)}/F^{(2)}$.

For the Marvin integrals one often leaves only a single free parameter $\mathcal{M}^{(0)}$, and give values to $\mathcal{M}^{(2)}$ and $\mathcal{M}^{(4)}$, by fixing the ratios $\mathcal{M}^{(2)}/\mathcal{M}^{(0)}$ and $\mathcal{M}^{(4)}/\mathcal{M}^{(0)}$.

For the pseudo-magnetic parameters again the common practice is to only leave a single free parameter $P^{(2)}$, and give values to $P^{(4)}$ and $P^{(6)}$, by fixing the ratios $P^{(4)}/P^{(2)}$ and $P^{(6)}/P^{(2)}$.

The values for all these ratios were historically obtained by using the integral expressions for the corresponding parameters, and calculating them using Hartree-Fock solutions to the radial parts of the wavefunctions. Examples of these ratios can be seen in the sections with data for LaF_3 and $LiYF_4$.

8 Fitting experimental data

qlanth also has the capacity to fit the semi-empirical Hamiltonian to experimental data. This is included in the sub-module `fittings.m` (see Appendix 18.2). This sub-module includes the function `ClassicalFit` which uses a truncated Hamiltonian (based on free-ion energies) to fit a given subset of the model parameters to given experimental data. It yields an extensive set of results, including fitted parameters and uncertainties. If the truncation energy parameter is set to infinity, then the fitting is performed with no truncation.

This function, however, is specifically used for fitting data for a single ion in a specific host. In the case of fitting data for several ions, it may be necessary to use parameter trends $\mathcal{P}(n)$. This is necessary since not only there might be some ions where there is no data (and where one would then propose a "synthetic" solution), but also since there are cases where there are too few data points to justify varying all of the model parameters.[23]

In these cases where one is fitting data for all or most of the lanthanide ions in a given host, it is useful to first fit the model in cases where there are the most data points, and to build up a parameter model $\mathcal{P}(n)$ for each parameter as the fitting of all the ions progresses. One feature often used in fitting for several ions (see Carnall *et al.* [Car+89] and Cheng *et al.* [Che+16]) is that when there is scarcity of data (as mentioned above), one can then use the trends in the $\mathcal{P}(n)$ in order to fix some parameter values at a given column (and proceed to vary others to fit the data to the model).[24]

Here below is a detailed explanation of the parameters required by `ClassicalFit`. The code for this function `ClassicalFit` may be found in Appendix 18.2.

- `numE`: number of electrons in the system, specifying the electronic configuration.

[23] The extreme case of this scarcity being Yb and Ce, where there are only 7 non-degenerate energies, but where the crystal-field alone might require more parameters than these (for instance in C_{2v} symmetry one needs 9 $B_q^{(k)}$ parameters). [24] For example, in the Table **??** for LaF_3, in the case of Yb, only two parameters are varied (ζ and ϵ) and the values for the crystal field obtained from linear fits to the previously fitted $B_q^{(k)}$ in Pr, Nd, Dy, Sm*, Ho*, Er, and Tm. (* not all $B_q^{(k)}$)

- `expData`: experimental data, a list of lists where each sublist represents an energy level and associated parameters. The first element of the sublists must represent energies, the other elements in the sublists are ignored but can be given to be kept together with the fitted data. The data must be ordered in increasing order of energy. **IMPORTANT:** if there are known unknown levels, these should be made explicit, anything other than a number will be interpreted as a level of undetermined energy in the corresponding gap. **ALSO IMPORTANT:** in the case of odd electron cases, `expData` needs to explicitly include the duplicate energies corresponding to Kramers' degeneracy; the gaps also need to be adequately duplicated in these cases.

- `excludeDataIndices`: indices in `expData` to be excluded from the fitting process. This can be used to exclude experimental data which is present, but which is considered dubious. In the case of odd electron configurations, these indices need to explicitly include the double degeneracy of Kramers doublets.

- `problemVars`: symbols representing the parameters to be fitted, some of which may be constrained (set fixed or proportional to others). **IMPORTANT:** if `problemVars` is a proper subset of all the parameters needed to evaluate the simplified Hamiltonian, the values for the other necessary parameters are taken from the Carnall *et al.* [Car+89] systematic study of LaF_3.

- `startValues`: an association with the initial values for the independent parameters given in `problemVars`. Independent parameters are those that remain once the constraints have been accounted for.

- σ`exp`: estimated uncertainty in the energy level differences between experimental and calculated values.

- `constraints`: a list of replacement rules defining constraints on the parameters. These constraints can either pin down a value, or apply proportionality ratios between them. If constrained by proportionally factors, these ratios are usually taken from Hartree-Fock calculations.

Here is a description of the different steps that this algorithm implements.

1. **Initialization**: sets initial conditions, processes options, and prepares data structures. Manages settings like the truncation energy, logging preferences, and computational accuracy goals.

2. **Data Preparation**: determines valid data points, excluding specified indices, and establishes truncation energy for the model.

3. **Hamiltonian Assembly and Simplification**: constructs the Hamiltonian while preserving its block structure, applies simplification rules, and processes the diagonal blocks to retain only free-ion parameters.

4. **Level Calculation**: determines the level description using free-ion parameters.

5. **Compilation and Truncation of Hamiltonian**: compiles the Hamiltonian and truncates it based on the set truncation energy, optimizing for computational efficiency.

6. **Fitting Process Initialization**: prepares variables and functions for optimization, including eigenvalue calculations and difference evaluations.

7. **Optimization**: employs the Levenberg-Marquardt method to optimize parameters, minimizing the discrepancy between calculated and experimental energy levels.

8. **Post-Processing**: calculates the Hamiltonian's eigensystem at the solution, deriving statistics like RMS deviation, parameter uncertainties, and covariance matrix.

9. **Output Compilation**: aggregates all relevant data and results into the output association `solCompendium`, documenting the fitting process and outcomes.

10. **Logging and Return**: saves the comprehensive fitting results to a log file and returns the detailed output data.

This function admits several options. Importantly here one may permit the model to have a constant shift to all the levels and the truncation energy can be set. Here one can also provide simplification rules that are applied to the compiled version of the Hamiltonian.

- `Energy Uncertainty in K`: used for error estimation, it can be either `Automatic` in which case the (σ of the fit is used as the uncertainty of the energies) or a numeric value in which case that is the value used for error propagation.
- `TruncationEnergy`: determines the energy level at which the Hamiltonian is truncated. If set to `Automatic`, the truncation energy is derived from the maximum energy present in the experimental data (`expData`). Otherwise, it can be manually set to a specific value.
- `MagneticSimplifier`: provides a list of replacement rules to simplify the magnetic parameters in the Hamiltonian, aiding in the reduction of computational complexity.
- `MagFieldSimplifier`: offers a list of replacement rules to specify a magnetic field, enhancing the flexibility in modeling magnetic effects within the system.
- `SymmetrySimplifier`: A list of replacement rules used to simplify the crystal field components of the Hamiltonian, facilitating a more efficient fitting process.
- `OtherSimplifier`: an additional list of replacement rules applied to the Hamiltonian before computation, allowing for further customization and simplification of the model, such as disabling specific interactions or effects. **IMPORTANT**: here the default is that the spin-spin contribution (as controlled by the σSS parameter) for the Marvin integrals is *not* included.
- `MaxHistory`: this option controls the length of the logs for the solver, enabling users to adjust the amount of log data retained during the fitting process.
- `MaxIterations`: sets the maximum number of iterations that the fitting algorithm (`NMinimize`) will execute, allowing control over the computational effort spent on the fitting.
- `FilePrefix`: specifies the prefix for the filenames under which the fitting results are saved. By default, the prefix is set to "calcs", and the files are saved in the "log/calcs" directory.
- `AddConstantShift`: if set to `True`, this option allows for a constant shift in the energy levels during the fitting process. This is particularly useful for fine-tuning the model to better match experimental data.
- `AccuracyGoal`: defines the accuracy goal for the `NMinimize` function used in the fitting process, allowing users to set the desired level of precision for the fit.
- `PrintFun`: specifies the function used to print progress messages during the fitting process. The default is `PrintTemporary`, which displays temporary output that can be useful for monitoring the fitting's progress.
- `SlackChannel`: names the Slack channel to which progress messages will be sent. If set to `None`, this feature is disabled, and no messages are sent to Slack.
- `ProgressView`: controls whether a progress window is displayed during the fitting process. When set to `True`, it provides an auxiliary notebook is created automatically with plots showing the progress of `NMinimize`.
- `SignatureCheck`: if `True`, the function ends prematurely and prints the list of the symbols that define the Hamiltonian after all basic simplifications have been applied without considering the given constraints.
- `SaveEigenvectors`: determines whether both the eigenvectors and eigenvalues of the fitted model are saved. If set to `False`, only the energies are saved.
- `AppendToLogFile`: what is provided here is appended to the log file under the "Appendix" key, enabling additional data to be stored alongside the fitting results.

The function returns an association with the following keys.

- `bestRMS`: the best root mean square deviation found during the fitting process.
- `bestParams`: the optimal set of parameters found through the fitting process.
- `paramSols`: a list of the parameter solutions at each step of the fitting algorithm.
- `timeTaken/s`: the total time taken to complete the fitting process, measured in seconds.
- `simplifier`: the replacement rules used to reduce the define the free-ion Hamiltonian.
- `excludeDataIndices`: the indices that were excluded from the fitting process as specified in the input.
- `startValues`: the initial values for the problem variables as given in the input.
- `freeIonSymbols`: symbols used in the intermediate coupling level calculation.
- `truncationEnergy`: the energy level at which the Hamiltonian was truncated.
- `numE`: the number of electrons in the f^{numE} configuration.
- `expData`: the experimental data used for the fitting process.
- `problemVars`: the variables considered during the fitting process.
- `maxIterations`: the maximum number of iterations used in the fitting process.
- `hamDim`: the dimension of the full Hamiltonian before simplifications or truncations.
- `allVars`: all the symbols defining the Hamiltonian under the applied simplifications.
- `freeBies`: the free-ion parameters used to calculate the intermediate coupling levels.
- `truncatedDim`: the dimension of the truncated Hamiltonian.
- `compiledIntermediateFname`: the file name of the compiled function used for the truncated Hamiltonian.
- `fittedLevels`: the number of levels that were fitted.
- `actualSteps`: the actual number of steps taken by the fitting algorithm.
- `solWithUncertainty`: a list of replacement rules showing the best fit value and its uncertainty for each parameter.
- `rmsHistory`: as list of the RMS values found during the fitting process.
- `Appendix`: an association appended to the log file under the "Appendix" key.
- `presentDataIndices`: the indices in `expData` that were used for fitting.
- `states`: a list of eigenvalues and eigenvectors for the fitted model, available if eigenvectors were saved.
- `energies`: a list of the energies of the fitted levels, adjusted if an energy shift was included in the fitting.

```
1 ClassicalFit::usage="ClassicalFit[numE, expData, excludeDataIndices,
    problemVars, startValues, constraints] fits the given expData in
    an f^numE configuration, by using the symbols in problemVars. The
    symbols given in problemVars may be constrained or held constant,
    this being controlled by constraints list which is a list of
    replacement rules expressing desired constraints. The constraints
    list additional constraints imposed upon the model parameters that
     remain once other simplifications have been ''baked'' into the
    compiled Hamiltonians that are used to increase the speed of the
    calculation.
2
```

```
3 Important, note that in the case of odd number of electrons the given
      data must explicitly include the Kramers degeneracy;
    excludeDataIndices must be compatible with this.
4
5 The list expData needs to be a list of lists with the only
    restriction that the first element of them corresponds to energies
     of levels. In this list, an empty value can be used to indicate
    known gaps in the data. Even if the energy value for a level is
    known (and given in expData) certain values can be omitted from
    the fitting procedure through the list excludeDataIndices, which
    correspond to indices in expData that should be skipped over.
6
7 The Hamiltonian used for fitting is a version that has been truncated
     either by using the maximum energy given in expData or by
    manually setting a truncation energy using the option ''
    TruncationEnergy''.
8
9 The option ''Experimental Uncertainty in K'' is the estimated
    uncertainty in the differences between the calculated and the
    experimental energy levels. This is used to estimate the
    uncertainty in the fitted parameters. Admittedly this will be a
    rough estimate (at least on the contribution of the calculated
    uncertainty), but it is better than nothing and may at least
    provide a lower bound to the uncertainty in the fitted parameters.
     It is assumed that the uncertainty in the differences between the
     calculated and the experimental energy levels is the same for all
     of them.
10
11 The list startValues is a list with all of the parameters needed to
    define the Hamiltonian (including the initial values for
    problemVars).
12
13 The function saves the solution to a file. The file is named with a
    prefix (controlled by the option ''FilePrefix'') and a UUID. The
    file is saved in the log sub-directory as a .m file.
14
15 Here's a description of the different parts of this function: first
    the Hamiltonian is assembled and simplified using the given
    simplifications. Then the intermediate coupling basis is
    calculated using the free-ion parameters for the given lanthanide.
     The Hamiltonian is then changed to the intermediate coupling
    basis and truncated. The truncated Hamiltonian is then compiled
    into a function that can be used to calculate the energy levels of
     the truncated Hamiltonian. The function that calculates the
    energy levels is then used to fit the experimental data. The
    fitting is done using FindMinimum with the Levenberg-Marquardt
    method.
16
17 The function returns an association with the following keys:
18
19 - ''bestRMS'' which is the best \[Sigma] value found.
20 - ''bestParams'' which is the best set of parameters found for the
    variables that were not constrained.
21 - ''bestParamsWithConstraints'' which has the best set of parameters
    (from - ''bestParams'') together with the used constraints. These
    include all the parameters in the model, even those that were not
    fitted for.
22 - ''paramSols'' which is a list of the parameters trajectories during
     the stepping of the fitting algorithm.
23 - ''timeTaken/s'' which is the time taken to find the best fit.
24 - ''simplifier'' which is the simplifier used to simplify the
    Hamiltonian.
25 - ''excludeDataIndices'' as given in the input.
26 - ''startValues'' as given in the input.
27
28 - ''freeIonSymbols'' which are the symbols used in the intermediate
    coupling basis.
29 - ''truncationEnergy'' which is the energy used to truncate the
    Hamiltonian, if it was set to Automatic, the value here is the
    actual energy used.
30 - ''numE'' which is the number of electrons in the f^numE
    configuration.
31 - ''expData'' which is the experimental data used for fitting.
32 - ''problemVars'' which are the symbols considered for fitting
33
34 - ''maxIterations'' which is the maximum number of iterations used by
```

```
    NMinimize.
- ''hamDim'' which is the dimension of the full Hamiltonian.
- ''allVars'' which are all the symbols defining the Hamiltonian
    under the aggregate simplifications.
- ''freeBies'' which are the free-ion parameters used to define the
    intermediate coupling basis.
- ''truncatedDim'' which is the dimension of the truncated
    Hamiltonian.
- ''compiledIntermediateFname'' the file name of the compiled
    function used for the truncated Hamiltonian.

- ''fittedLevels'' which is the number of levels fitted for.
- ''actualSteps'' the number of steps that FindMiniminum actually
    took.
- ''solWithUncertainty'' which is a list of replacement rules of the
    form (paramSymbol -> {bestEstimate, uncertainty}).
- ''rmsHistory'' which is a list of the \[Sigma] values found during
    the fitting.
- ''Appendix'' which is an association appended to the log file under
     the key ''Appendix''.
- ''presentDataIndices'' which is the list of indices in expData that
     were used for fitting, this takes into account both the empty
    indices in expData and also the indices in excludeDataIndices.

- ''states'' which contains a list of eigenvalues and eigenvectors
    for the fitted model, this is only available if the option ''
    SaveEigenvectors'' is set to True; if a general shift of energy
    was allowed for in the fitting, then the energies are shifted
    accordingly.
- ''energies'' which is a list of the energies of the fitted levels,
    this is only available if the option ''SaveEigenvectors'' is set
    to False. If a general shift of energy was allowed for in the
    fitting, then the energies are shifted accordingly.
- ''degreesOfFreedom'' which is equal to the nunber of fitted state
    energies minus the number of parameters used in fitting.

The function admits the following options with corresponding default
    values:
- ''MaxHistory'' : determines how long the logs for the solver can be
    .
- ''MaxIterations'': determines the maximum number of iterations used
     by NMinimize.
- ''FilePrefix'' : the prefix to use for the subfolder in the log
    filder, in which the solution files are saved, by default this is
    ''calcs'' so that the calculation files are saved under the
    directory ''log/calcs''.
- ''AddConstantShift'' : if True then a constant shift is allowed in
    the fitting, default is False. If this is the case the variable
    ''\[Epsilon]'' is added to the list of variables to be fitted for,
     it must not be included in problemVars.

- ''AccuracyGoal'': the accuracy goal used by NMinimize, default of
    5.
- ''TrucationEnergy'': if Automatic then the maximum energy in
    expData is taken, else it takes the value set by this option. In
    all cases the energies in expData are only considered up to this
    value.
- ''PrintFun'': the function used to print progress messages, the
    default is PrintTemporary.
- ''RefParamsVintage'': the vintage of the reference parameters to
    use. The reference parameters are both used to determine the
    truncated Hamiltonian, and also as starting values for the solver.
     It may be ''LaF3'', in which case reference parameters from
    Carnall are used. It may also be ''LiYF4'', in which case the
    reference parameters from the LiYF4 paper are used. It may also be
     Automatic, in which case the given experimental data is used to
    determine starting values for F^k and ζ. It may also be a list or
    association that provides values for the Slater integrals and spin
    -orbit coupling, the remaining necessary parameters complemented
    by using ''LaF3''.

- ''ProgressView'': whether or not a progress window will be opened
    to show the progress of the solver, the default is True.
- ''SignatureCheck'': if True then then the function returns
    prematurely, returning a list with the symbols that would have
    defined the Hamiltonian after all simplifications have been
```

```
    applied. Useful to check the entire parameter set that the
    Hamiltonian has, which has to match one-to-one what is provided by
     startValues.
- ''SaveEigenvectors'': if True then the both the eigenvectors and
    eigenvalues are saved under the ''states'' key of the returned
    association. If False then only the energies are saved, the
    default is False.

- ''AppendToLogFile'': an association appended to the log file under
    the key ''Appendix''.
- ''MagneticSimplifier'': a list of replacement rules to simplify the
     Marvin and pesudo-magnetic paramters. Here the ratios of the
    Marvin parameters and the pseudo-magnetic parameters are defined
    to simplify the magnetic part of the Hamiltonian.
- ''MagFieldSimplifier'': a list of replacement rules to specify a
    magnetic field (in T), if set to {}, then {Bx, By, Bz} can also be
     used as variables to be fitted for.

- ''SymmetrySimplifier'': a list of replacements rules to simplify
    the crystal field.
- ''OtherSimplifier'': an additional list of replacement rules that
    are applied to the Hamiltonian before computing with it. Here the
    spin-spin contribution can be turned off by setting \[Sigma]SS->0,
     which is the default.
";
Options[ClassicalFit] = {
  "MaxHistory"       -> 200,
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "calcs",
  "ProgressView"     -> True,
  "TruncationEnergy" -> Automatic,
  "AccuracyGoal"     -> 5,
  "PrintFun"         -> PrintTemporary,
  "RefParamsVintage" -> "LaF3",
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "SaveToLog"        -> False,
  "Energy Uncertainty in K" -> Automatic,
  "MagneticSimplifier" -> {
    M2 -> 56/100 M0,
    M4 -> 31/100 M0,
    P4 -> 1/2 P2,
    P6 -> 1/10 P2
    },
  "MagFieldSimplifier" -> {
    Bx -> 0,
    By -> 0,
    Bz -> 0
    },
  "SymmetrySimplifier" -> {
    B12->0, B14->0, B16->0, B34->0, B36->0, B56->0,
    S12->0 ,S14->0, S16->0, S22->0, S24->0, S26->0,
    S34->0 ,S36->0, S44->0, S46->0, S56->0, S66->0
    },
  "OtherSimplifier" -> {
    F0->0,
    P0->0,
    \[Sigma]SS->0,
    T11p->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0,
    wChErrA ->0, wChErrB->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19
    ->0,
      T2p->0},
    2 -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19
    ->0,
      T2p->0
      },
```

```
    3  -> {},
    4  -> {},
    5  -> {},
    6  -> {},
    7  -> {},
    8  -> {},
    9  -> {},
    10 -> {},
    11 -> {},
    12 -> {
      T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19
    ->0,
      T2p->0
      },
    13->{
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19
    ->0,
      T2p->0
      }
    |>,
    "FreeIonSymbols" -> {F0, F2, F4, F6, ζ}
};
ClassicalFit[numE_Integer, expData_List, excludeDataIndices_List,
    problemVars_List, startValues_Association, constraints_List,
    OptionsPattern[]]:=Module[
  {
    accuracyGoal,
    allFreeEnergies,
    allFreeEnergiesSorted, allVars, allVarsVec,
    argsForEvalInsideOfTheIntermediateSystems,
    argsOfTheIntermediateEigensystems, aVar, aVarPosition,
    basis, basisChanger, basisChangerBlocks,
    bestParams, bestRMS, blockShifts, blockSizes,
    compiledDiagonal, compiledIntermediateFname,
    constrainedProblemVars, constrainedProblemVarsList,
    currentRMS, degressOfFreedom, dependentVars,
    diagonalBlocks, diagonalScalarBlocks, diff,
    eigenEnergies, eigenvalueDispenserTemplate,
    eigenVectors, elevatedIntermediateEigensystems,
    endTime, fmSolAssoc, freeBies,
    freeIenergiesAndMultiplets, fullHam, fullSolVec,
    ham, hamDim, hamEigenvaluesTemplate,
    hamString, indepSolVecVec, indepVars, intermediateHam,
    isolationValues, lin, linMat, ln, lnParams,
    logFilePrefix, magneticSimplifier,
    maxFreeEnergy, maxHistory, maxIterations,
    minFreeEnergy, minpoly,
    modelSymbols, multipletAssignments, needlePosition,
    numBlocks, solCompendium,
    openNotebooks, ordering, otherSimplifier, p0,
    paramBest, perHam,
    presentDataIndices, PrintFun, problemVarsPositions,
    problemVarsQ, problemVarsQString, problemVarsVec,
    problemVarsWithStartValues, reducedModelSymbols,
    roundedTruncationEnergy,
    runningInteractive, shiftToggle, simplifier,
    sol, solWithUncertainty,
    sortedTruncationIndex, sqdiff, standardValues,
    startTime,
    states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, timeTaken,
    truncatedIntermediateBasis, truncatedIntermediateHam,
    truncationEnergy, truncationIndices, RefParams,
    truncationUmbral, varHash,
    varsWithConstants, \[Lambda]0Vec,
    \[Lambda]exp
    },
  (
    \[Sigma]exp   = OptionValue["Energy Uncertainty in K"];
    solCompendium = <||>;
    refParamsVintage = OptionValue["RefParamsVintage"];
    RefParams = Which[
      refParamsVintage === "LaF3",
```

```
195       LoadLaF3Parameters,
196       refParamsVintage === "LiYF4",
197       LoadLiYF4Parameters,
198       True,
199       refParamsVintage
200     ];
201     hamDim         = Binomial[14, numE];
202     addShift       = OptionValue["AddConstantShift"];
203     ln             = theLanthanides[[numE]];
204     maxHistory     = OptionValue["MaxHistory"];
205     maxIterations  = OptionValue["MaxIterations"];
206     logFilePrefix  = If[OptionValue["FilePrefix"] == "",
207                         ToString[theLanthanides[[numE]]],
208                         OptionValue["FilePrefix"]
209                       ];
210     accuracyGoal   = OptionValue["AccuracyGoal"];
211     PrintFun       = OptionValue["PrintFun"];
212     freeIonSymbols       = OptionValue["FreeIonSymbols"];
213     runningInteractive   = (Head[$ParentLink] === LinkObject);
214     magneticSimplifier   = OptionValue["MagneticSimplifier"];
215     magFieldSimplifier   = OptionValue["MagFieldSimplifier"];
216     symmetrySimplifier   = OptionValue["SymmetrySimplifier"];
217     otherSimplifier      = OptionValue["OtherSimplifier"];
218     threeBodySimplifier  = If[Head[OptionValue["ThreeBodySimplifier"]]
        == Association,
219                             OptionValue["ThreeBodySimplifier"][numE],
220                             OptionValue["ThreeBodySimplifier"]
221                           ];
222
223     truncationEnergy = If[OptionValue["TruncationEnergy"] ===
    Automatic,
224       (
225         PrintFun["Truncation energy set to Automatic, using the
    maximum energy (+20%) in the data ..."];
226         Round[1.2 * Max[Select[First /@ expData, NumericQ[#] &]]]
227       ),
228       OptionValue["TruncationEnergy"]
229       ];
230     truncationEnergy = Max[50000, truncationEnergy];
231     PrintFun["Using a truncation energy of ", truncationEnergy, " K"
    ];
232
233     simplifier = Join[magneticSimplifier,
234                       magFieldSimplifier,
235                       symmetrySimplifier,
236                       threeBodySimplifier,
237                       otherSimplifier];
238
239     PrintFun["Determining gaps in the data ..."];
240     (* whatever is non-numeric is assumed as a known gap *)
241     presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
    , ___}]];
242     (* some indices omitted here based on the excludeDataIndices
    argument *)
243     presentDataIndices = Complement[presentDataIndices,
    excludeDataIndices];
244
245     solCompendium["simplifier"]          = simplifier;
246     solCompendium["excludeDataIndices"] = excludeDataIndices;
247     solCompendium["startValues"]         = startValues;
248     solCompendium["freeIonSymbols"]      = freeIonSymbols;
249     solCompendium["truncationEnergy"]    = truncationEnergy;
250     solCompendium["numE"]                = numE;
251     solCompendium["expData"]             = expData;
252     solCompendium["problemVars"]         = problemVars;
253     solCompendium["maxIterations"]       = maxIterations;
254     solCompendium["hamDim"]              = hamDim;
255     solCompendium["constraints"]         = constraints;
256
257     modelSymbols   = Sort[Select[paramSymbols, Not[MemberQ[Join[
    racahSymbols, juddOfeltIntensitySymbols, chenSymbols,{t2Switch, \[
    Epsilon], gs, nE}],#]]&]];
258     (* remove the symbols that will be removed by the simplifier, no
    symbol should remain here that is not in the symbolic Hamiltonian
    *)
259     reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
```

```
    simplifier],#]]&];

    (* this is useful to understand what are the arguments of the
    truncated compiled Hamiltonian *)
    If[OptionValue["SignatureCheck"],
      (
        PrintFun["Given the model parameters and the simplifying
    assumptions, the resultant model parameters are:"];
        PrintFun[{reducedModelSymbols}];
        PrintFun["Exiting ..."];
        Return[""];
      )
    ];

    (* calculate the basis *)
    PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
    basis    = BasisLSJMJ[numE];

    Which[refParamsVintage === Automatic,
      (
        PrintFun["Using the automatic vintage with freshly fitted
    free-ion parameters and others as in LaF3 ..."];
        lnParams      = LoadLaF3Parameters[ln];
        freeIonSol    = FreeIonSolver[expData, numE];
        freeIonParams = freeIonSol["bestParams"];
        lnParams      = Join[lnParams, freeIonParams];
      ),
      MemberQ[{List, Association}, Head[RefParams]],
      (
        RefParams = Association[RefParams];
        PrintFun["Using the given parameters as a starting point ..."
    ];
        lnParams    = RefParams;
        extraParams = LoadLaF3Parameters[ln];
        lnParams    = Join[extraParams, lnParams];
      ),
      True,
      (
        (* get the reference parameters from the given vintage *)
        PrintFun["Getting reference free-ion parameters for ", ln, "
    using ", refParamsVintage, " ..."];
        lnParams = ParamPad[RefParams[ln], "PrintFun" -> PrintFun];
      )
    ];
    freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
      numE];
    (* a more explicit alias *)
    allVars            = reducedModelSymbols;
    numericConstraints = Association@Select[constraints, NumericQ
    [#[[2]]] &];
    standardValues     = allVars /. Join[lnParams, numericConstraints
    ];
    solCompendium["allVars"]  = allVars;
    solCompendium["freeBies"] = freeBies;

    (* reload compiled version if found *)
    varHash                   = Hash[{numE, allVars, freeBies,
    truncationEnergy, simplifier}];
    compiledIntermediateFname = ln <> "-compiled-intermediate-
    truncated-ham-" <> ToString[varHash] <> ".mx";
    compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
    compiledIntermediateFname}];
    solCompendium["compiledIntermediateFname"] =
    compiledIntermediateFname;

    If[FileExistsQ[compiledIntermediateFname],
      PrintFun["This ion, free-ion params, and full set of variables
    have been used before (as determined by {numE, allVars, freeBies,
    truncationEnergy, simplifier}). Loading the previously saved
    compiled function and intermediate coupling basis ..."];
      PrintFun["Using : ", compiledIntermediateFname];
      {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
     Import[compiledIntermediateFname];,
    (
      If[truncationEnergy == Infinity,
      (
```

```
319           ham = EffectiveHamiltonian[numE, "ReturnInBlocks" -> False
    ];
320           theSimplifier = simplifier;
321           ham = Normal@ReplaceInSparseArray[ham, simplifier];
322           PrintFun["Compiling a function for the Hamiltonian with no
    truncation ..."];
323           (* compile a function that will calculate the truncated
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
324           compileIntermediateTruncatedHam = Compile[Evaluate[allVars
    ], Evaluate[ham]];
325           truncatedIntermediateBasis = SparseArray@IdentityMatrix[
    Binomial[14, numE]];
326           (* save the compiled function *)
327           PrintFun["Saving the compiled function for the Hamiltonian
    with no truncation and a placeholder intermediate basis ..."];
328           Export[compiledIntermediateFname, {
    compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
329       ),
330       (
331         (* grab the Hamiltonian preserving the block structure *)
332         PrintFun["Assembling the Hamiltonian for f^",numE," keeping
    the block structure ..."];
333         ham           = EffectiveHamiltonian[numE, "ReturnInBlocks"->
    True];
334         (* apply the simplifier *)
335         PrintFun["Simplifying using the aggregate set of
    simplification rules ..."];
336         ham           = Map[ReplaceInSparseArray[#, simplifier]&, ham,
     {2}];
337         PrintFun["Zeroing out every symbol in the Hamiltonian that is
     not a free-ion parameter ..."];
338         (* Get the free ion symbols *)
339         freeIonSimplifier = (#->0) & /@ Complement[
    reducedModelSymbols, freeIonSymbols];
340         (* Take the diagonal blocks for the intermediate analysis *)
341         PrintFun["Grabbing the diagonal blocks of the Hamiltonian ...
    "];
342         diagonalBlocks       = Diagonal[ham];
343         (* simplify them to only keep the free ion symbols *)
344         PrintFun["Simplifying the diagonal blocks to only keep the
    free ion symbols ..."];
345         diagonalScalarBlocks = ReplaceInSparseArray[#,
    freeIonSimplifier]&/@diagonalBlocks;
346         (* these include the MJ quantum numbers, remove that *)
347         PrintFun["Contracting the basis vectors by removing the MJ
    quantum numbers from the diagonal blocks ..."];
348         diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
349
350         argsOfTheIntermediateEigensystems         = StringJoin[Riffle
    [Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
351         argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle
    [(ToString[#]<>"v") & /@ freeIonSymbols,", "]];
352         PrintFun["argsOfTheIntermediateEigensystems = ",
    argsOfTheIntermediateEigensystems];
353         PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
    argsForEvalInsideOfTheIntermediateSystems];
354         PrintFun["(if the following fails, it might help to see if
    the arguments of TheIntermediateEigensystems match the ones shown
    above)"];
355
356         (* compile a function that will effectively calculate the
    spectrum of all of the scalar blocks given the parameters of the
    free-ion part of the Hamiltonian *)
357         (* compile one function for each of the blocks *)
358         PrintFun["Compiling functions for the diagonal blocks of the
    Hamiltonian ..."];
359         compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate
    [N[Normal[#]]]]&/@diagonalScalarBlocks;
360         (* use that to create a function that will calculate the free
    -ion eigensystem *)
361         TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζ
    v_] := (
362           theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&) /@
    compiledDiagonal;
363           theIntermediateEigensystems = Eigensystem /@
```

```
    theNumericBlocks;
364         Js     = AllowedJ[numEv];
365         basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
366         (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
367         elevatedIntermediateEigensystems = MapIndexed[EigenLever
    [#1, 2Js[[#2[[1]]]]+1 ]&, theIntermediateEigensystems];
368         (* Identify a single MJ to keep *)
369         pivot          = If[EvenQ[numEv],0,-1/2];
370         LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
371         (* calculate the multiplet assignments that the
    intermediate basis eigenvectors have *)
372         needlePosition = 0;
373         multipletAssignments = Table[
374           (
375             J          = Js[[idx]];
376             eigenVecs = theIntermediateEigensystems[[idx]][[2]];
377             majorComponentIndices         = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
378             majorComponentIndices       += needlePosition;
379             needlePosition              += Length[
    majorComponentIndices];
380             majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
381             (* All of the degenerate eigenvectors belong to the
    same multiplet*)
382             elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments, 2J+1];
383             elevatedMultipletAssignments
384           ),
385         {idx, 1, Length[Js]}
386         ];
387         (* put together the multiplet assignments and the energies
    *)
388         freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
389         freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
390         (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
391         basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
392         basisChanger = SparseArray[basisChanger];
393         Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
394       );
395
396       PrintFun["Calculating the intermediate eigensystems for ",ln,
    " using free-ion params from LaF3 ..."];
397       (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
398       {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
399
400       (* use that intermediate coupling basis to compile a function
     for the full Hamiltonian *)
401       allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
402       (* important that the intermediate coupling basis have
    attached energies, which make possible the truncation *)
403       ordering = Ordering[allFreeEnergies];
404       (* sort the free ion energies and determine which indices
    should be included in the truncation *)
405       allFreeEnergiesSorted          = Sort[allFreeEnergies];
406       {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
407       (* determine the index at which the energy is equal or larger
     than the truncation energy *)
408       sortedTruncationIndex = Which[
409         truncationEnergy > (maxFreeEnergy-minFreeEnergy),
410         hamDim,
411         True,
412         FirstPosition[allFreeEnergiesSorted - Min[
    allFreeEnergiesSorted],x_/;x>truncationEnergy,{0},1][[1]]
```

```
413         ];
414         (* the actual energy at which the truncation is made *)
415         roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
416 
417         (* the indices that participate in the truncation *)
418         truncationIndices = ordering[[;;sortedTruncationIndex]];
419         PrintFun["Computing the block structure of the change of
    basis array ..."];
420         blockSizes          = BlockArrayDimensionsArray[ham];
421         basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
422         blockShifts         = First /@ Diagonal[blockSizes];
423         numBlocks           = Length[blockSizes];
424         (* using the ham (with all the symbols) change the basis to
    the computed one *)
425         PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
426         intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks
    ];
427         PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
428         Do[
429           intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
430           {rowIdx, 1, numBlocks},
431           {colIdx, 1, numBlocks}
432         ];
433         intermediateHam = BlockMatrixMultiply[BlockTranspose[
    basisChangerBlocks],intermediateHam];
434         PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
435         Do[
436           intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
437           {rowIdx, 1, numBlocks},
438           {colIdx, 1, numBlocks}
439         ];
440         (* using the truncation indices truncate that one *)
441         PrintFun["Truncating the Hamiltonian ..."];
442         truncatedIntermediateHam = TruncateBlockArray[intermediateHam
    , truncationIndices, blockShifts];
443         (* these are the basis vectors for the truncated hamiltonian
    *)
444         PrintFun["Saving the truncated intermediate basis ..."];
445         truncatedIntermediateBasis = basisChanger[[All,
    truncationIndices]];
446 
447         PrintFun["Compiling a function for the truncated Hamiltonian
    ..."];
448         (* compile a function that will calculate the truncated
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
449         compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
    Evaluate[truncatedIntermediateHam]];
450         (* save the compiled function *)
451         PrintFun["Saving the compiled function for the truncated
    Hamiltonian and the truncated intermediate basis ..."];
452         Export[compiledIntermediateFname, {
    compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
453       )
454     ];
455     )
456     ];
457 
458     truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
459     PrintFun["The truncated Hamiltonian has a dimension of ",
    truncationUmbral, "x", truncationUmbral, " ..."];
460     presentDataIndices = Select[presentDataIndices, # <=
    truncationUmbral &];
461     solCompendium["presentDataIndices"] = presentDataIndices;
462 
463     (* the problemVars are the symbols that will be fitted for *)
464 
465     PrintFun["Starting up the fitting process using the Levenberg-
    Marquardt method ..."];
466     (* using the problemVars I need to create the argument list
```

```
    including _?NumericQ *)
467 problemVarsQ       = (ToString[#] <> "_?NumericQ") & /@
    problemVars;
468 problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
469 (* we also need to have the padded arguments with the variables
    in the right position and the fixed values in the remaining ones
    *)
470 problemVarsPositions = Position[allVars, #][[1, 1]] & /@
    problemVars;
471 problemVarsString    = StringJoin[Riffle[ToString /@ problemVars,
     ", "]];
472 (* to feed parameters to the Hamiltonian, which includes all
    parameters, we need to form the set of arguments, with fixed
    values where needed, and the variables in the right position *)
473 varsWithConstants                          = standardValues;
474 varsWithConstants[[problemVarsPositions]] = problemVars;
475 varsWithConstantsString                    = ToString[
    varsWithConstants];
476
477 (* this following function serves eigenvalues from the
    Hamiltonian, has memoization so it might grow to use a lot of RAM
    *)
478 Clear[HamSortedEigenvalues];
479 hamEigenvaluesTemplate = StringTemplate["
480 HamSortedEigenvalues[`problemVarsQ`]:=(
481   ham         = compileIntermediateTruncatedHam@@`
    varsWithConstants`;
482   eigenValues = Chop[Sort@Eigenvalues@ham];
483   eigenValues = eigenValues - Min[eigenValues];
484   HamSortedEigenvalues[`problemVarsString`] = eigenValues;
485   Return[eigenValues]
486 )"];
487 hamString = hamEigenvaluesTemplate[<|
488     "problemVarsQ"      -> problemVarsQString,
489     "varsWithConstants" -> varsWithConstantsString,
490     "problemVarsString" -> problemVarsString
491     |>];
492 ToExpression[hamString];
493
494 (* we also need a function that will pick the i-th eigenvalue,
    this seems unnecessary but it's needed to form the right
    functional form expected by the Levenberg-Marquardt method *)
495 eigenvalueDispenserTemplate = StringTemplate["
496 PartialHamEigenvalues[`problemVarsQ`][i_]:=(
497   eigenVals = HamSortedEigenvalues[`problemVarsString`];
498   eigenVals[[i]]
499 )
500 "];
501 eigenValueDispenserString = eigenvalueDispenserTemplate[<|
502     "problemVarsQ"      -> problemVarsQString,
503     "problemVarsString" -> problemVarsString
504     |>];
505 ToExpression[eigenValueDispenserString];
506
507 PrintFun["Determining the free variables after constraints ..."];
508 constrainedProblemVars     = (problemVars /. constraints);
509 constrainedProblemVarsList = Variables[constrainedProblemVars];
510 If[addShift,
511   PrintFun["Adding a constant shift to the fitting parameters ...
    "];
512   constrainedProblemVarsList = Append[constrainedProblemVarsList,
     \[Epsilon]]
513 ];
514
515 indepVars = Complement[problemVars, #[[1]] & /@ constraints];
516 stringPartialVars = ToString/@constrainedProblemVarsList;
517
518 paramSols  = {};
519 rmsHistory = {};
520 steps      = 0;
521 problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
522 If[addShift,
523   problemVarsWithStartValues = Append[problemVarsWithStartValues,
     {\[Epsilon] ,0}];
524 ];
525 openNotebooks     = If[runningInteractive,
```

```
526             ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
    [],
527             {}];
528     If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
    ProgressView"],
529       ProgressNotebook["Basic"->False]
530     ];
531     degressOfFreedom = Length[presentDataIndices] - Length[
    problemVars] - 1;
532     PrintFun["Fitting for ", Length[presentDataIndices], " data
    points with ", Length[problemVars], " free parameters.", " The
    effective degrees of freedom are ", degressOfFreedom, " ..."];
533
534     PrintFun["Fitting model to data ..."];
535     startTime   = Now;
536     shiftToggle = If[addShift, 1, 0];
537     sol = FindMinimum[
538       Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@
    constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
539       {j, presentDataIndices}],
540       problemVarsWithStartValues,
541       Method        -> "LevenbergMarquardt",
542       MaxIterations -> OptionValue["MaxIterations"],
543       AccuracyGoal  -> OptionValue["AccuracyGoal"],
544       StepMonitor :> (
545         steps      += 1;
546         currentSqSum = Sum[(expData[[j]][[1]] - (
    PartialHamEigenvalues @@ constrainedProblemVars)[j] - shiftToggle
    * \[Epsilon])^2, {j, presentDataIndices}];
547         currentRMS = Sqrt[currentSqSum / degressOfFreedom];
548         paramSols  = AddToList[paramSols, constrainedProblemVarsList,
     maxHistory];
549         rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
550         )
551       ];
552     endTime = Now;
553     timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
554     PrintFun["Solution found in ", timeTaken, "s"];
555
556     solVec = constrainedProblemVars /. sol[[-1]];
557     indepSolVec = indepVars /. sol[[-1]];
558     If[addShift,
559       \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
560       \[Epsilon]Best = 0
561     ];
562     fullSolVec = standardValues;
563     fullSolVec[[problemVarsPositions]] = solVec;
564     PrintFun["Calculating the truncated numerical Hamiltonian
    corresponding to the solution ..."];
565     fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
566     PrintFun["Calculating energies and eigenvectors ..."];
567     {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
568     states = Transpose[{eigenEnergies, eigenVectors}];
569     states = SortBy[states, First];
570     eigenEnergies = First /@ states;
571     PrintFun["Shifting energies to make ground state zero of energy
    ..."];
572     eigenEnergies = eigenEnergies - eigenEnergies[[1]];
573     PrintFun["Calculating the linear approximant to each eigenvalue
    ..."];
574     allVarsVec = Transpose[{allVars}];
575     p0 = Transpose[{fullSolVec}];
576     linMat = {};
577     If[addShift,
578       tail = -2,
579       tail = -1];
580     Do[
581       (
582         aVarPosition = Position[allVars, aVar][[1, 1]];
583         isolationValues = ConstantArray[0, Length[allVars]];
584         isolationValues[[aVarPosition]] = 1;
585         dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
    constraints]];
586         Do[
587           isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
```

```
588             {dVar, dependentVars}
589           ];
590           perHam  = compileIntermediateTruncatedHam @@ isolationValues;
591           lin     = FirstOrderPerturbation[Last /@ states, perHam];
592           linMat  = Append[linMat, lin];
593         ),
594         {aVar, constrainedProblemVarsList[[;;tail]]}
595       ];
596       PrintFun["Removing the gradient of the ground state ..."];
597       linMat = (# - #[[1]] & /@ linMat);
598       PrintFun["Transposing derivative matrices into columns ..."];
599       linMat  = Transpose[linMat];
600
601       PrintFun["Calculating the eigenvalue vector at solution ..."];
602       \[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
603       PrintFun["Putting together the experimental vector ..."];
604       \[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
          ]]}];
605       problemVarsVec = If[addShift,
606         Transpose[{constrainedProblemVarsList[[;;-2]]}],
607         Transpose[{constrainedProblemVarsList}]
608       ];
609       indepSolVecVec = Transpose[{indepSolVec}];
610       PrintFun["Calculating the difference between eigenvalues at
          solution ..."];
611       diff = If[linMat == {},
612         (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
613         (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
          presentDataIndices]].(problemVarsVec - indepSolVecVec)
614       ];
615       PrintFun["Calculating the sum of squares of differences around
          solution ... "];
616       sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
617       PrintFun["Calculating the minimum (which should coincide with sol
          ) ..."];
618       minpoly       = sqdiff /. AssociationThread[problemVars -> solVec
          ];
619       fmSolAssoc    = Association[sol[[2]]];
620       If[\[Sigma]exp == Automatic,
621         \[Sigma]exp = Sqrt[minpoly / degressOfFreedom];
622       ];
623       \[CapitalDelta]\[Chi]2 = Sqrt[degressOfFreedom];
624       Amat = (1/\[Sigma]exp^2) * Transpose[linMat[[presentDataIndices
          ]]].linMat[[presentDataIndices]];
625       paramIntervals = EllipsoidBoundingBox[Amat, \[CapitalDelta]\[Chi
          ]2];
626       PrintFun["Calculating the uncertainty in the parameters ..."];
627       solWithUncertainty = Table[
628         (
629           aVar        = constrainedProblemVarsList[[varIdx]];
630           paramBest   = aVar /. fmSolAssoc;
631           (aVar -> {paramBest, paramIntervals[[varIdx, 2]]})
632         ),
633       {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
634       ];
635
636       bestRMS    = Sqrt[minpoly / degressOfFreedom];
637       bestParams = sol[[2]];
638       bestWithConstraints = Association@Join[constraints, bestParams];
639       bestWithConstraints = bestWithConstraints /. bestWithConstraints;
640       bestWithConstraints = (# + 0.) & /@ bestWithConstraints;
641
642       solCompendium["degreesOfFreedom"]    = degressOfFreedom;
643       solCompendium["solWithUncertainty"]  = solWithUncertainty;
644       solCompendium["truncatedDim"]        = truncationUmbral;
645       solCompendium["fittedLevels"]        = Length[presentDataIndices
          ];
646       solCompendium["actualSteps"]         = steps;
647       solCompendium["bestRMS"]             = bestRMS;
648       solCompendium["problemVars"]         = problemVars;
649       solCompendium["paramSols"]           = paramSols;
650       solCompendium["rmsHistory"]          = rmsHistory;
651       solCompendium["Appendix"]            = OptionValue["
          AppendToLogFile"];
652       solCompendium["timeTaken/s"]         = timeTaken;
653       solCompendium["bestParams"]          = bestParams;
```

```
    solCompendium["bestParamsWithConstraints"] = bestWithConstraints;

    If[OptionValue["SaveEigenvectors"],
        solCompendium["states"] = {#[[1]] + \[Epsilon]Best, #[[2]]}
    &/@ (Chop /@ ShiftedLevels[states]),
        (
          finalEnergies = Sort[First /@ states];
          finalEnergies = finalEnergies - finalEnergies[[1]];
          finalEnergies = finalEnergies + \[Epsilon]Best;
          finalEnergies  = Chop /@ finalEnergies;
          solCompendium["energies"] = finalEnergies;
      )
    ];
    If[OptionValue["SaveToLog"],
      PrintFun["Saving the solution to the log file ..."];
      LogSol[solCompendium, logFilePrefix];
    ];
    PrintFun["Finished ..."];
    Return[solCompendium];
  )
];
```

9 Accompanying notebooks

The code for this dissertation is accompanied by the following auxiliary *Mathematica* notebooks, which either document the functions included in the code, or serve as aids in the calculation of matrix elements. These notebooks assume that the `paclet` has been installed according to the instructions given in Section 16.

- 🖫 `/notebooks/qlanth - Table Generator.nb`: generates the basic tables on which every calculation is based. This means that LS-reduced matrix elements are used to calculate matrix elements in the $|LSJM_J\rangle$ basis.
- 🖫 `/notebooks/qlanth - JJBlock Calculator.nb`: can be used to generate the J-J' blocks for the different interactions. The data files produced here are necessary for `EffectiveHamiltonian` to work. These blocks are creating by putting together matrix elements from different interactions.
- 🖫 `/notebooks/The Lanthanides in LaF3.nb`: runs **`qlanth`** over the lanthanide ions in LaF_3 and compares the results against the published values from Carnall *et al.* [Car+89]. It also calculates magnetic dipole transition rates and oscillator strengths.

10 Compiled data for LaF_3:Ln^{3+} and $LiYF_4$:Ln^{3+}

The study of Carnall *et al.* [Car+89] on lanthanum fluoride was a systematic review of trivalent lanthanide ions in LaF_3. In this work they fitted the experimental data for all of the lanthanide ions using the single-configuration effective Hamiltonian. In their appendices one can find their calculated values, together with the experimental values that they used for their least squares fittings. In **`qlanth`** this data can be accessed by invoking the command `LoadCarnall` which brings into the session an association that has keys that have as values the tables and appendices from this article. Fig-6 shows the results of a calculation done with **`qlanth`** for the energy levels in LaF_3. Additionally the function `LoadLaF3Parameters` can be used to query the data for the fitted parameters, which may serve as a useful starting point for the description of the lanthanides ions in hosts other than LaF_3.

Similarly, Cheng *et al.* [Che+16] compiled data for $LiYF_4$. In **`qlanth`** model parameters for $LiYF_4$ can be obtained by calling the function `LoadLiYF4Parameters`. Fig-7 shows the results of a calculation done with **`qlanth`** for the energy levels in $LiYF_4$.

```
Carnall::usage = "Association of data from Carnall et al (1989) with
    the following keys: {data, annotations, paramSymbols, elementNames
    , rawData, rawAnnotations, annnotatedData, appendix:Pr:Association
    , appendix:Pr:Calculated, appendix:Pr:RawTable, appendix:Headings}
    ";
```

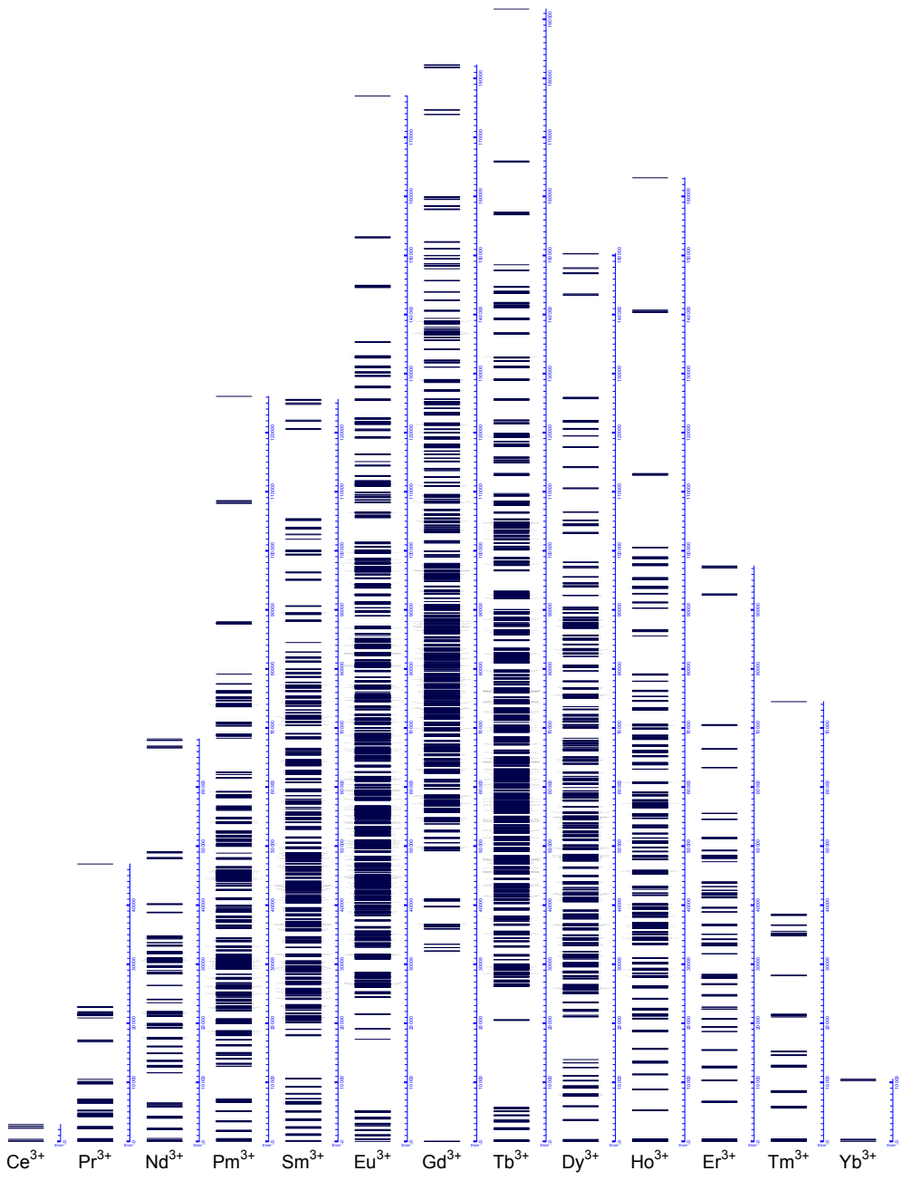


Figure 6: Energy levels in LaF_3.

```
LoadCarnall::usage = "LoadCarnall[] loads data for trivalent
    lanthanides in LaF3 using the data from Bill Carnall's 1989 paper.
    ";
LoadCarnall[] := (
  If[ValueQ[Carnall], Return[]];
  carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
  If[!FileExistsQ[carnallFname],
    (PrintTemporary[">> Carnall.m not found, generating ..."];
      Carnall = ParseCarnall[];
    ),
    Carnall = Import[carnallFname];
  ];
);
```

```
LoadLaF3Parameters::usage="LoadLaF4Parameters[ln] gives the models
    for the semi-empirical Hamiltonian for the trivalent lanthanide
    ion with symbol ln.";
Options[LoadLaF3Parameters] = {
  "Vintage"  -> "Standard",
  "With Uncertainties"->False
};
LoadLaF3Parameters[Ln_String, OptionsPattern[]]:= Module[
  {paramData, params},
  (
    paramData = Which[
      OptionValue["Vintage"] == "Carnall",
      Import[FileNameJoin[{moduleDir, "data", "carnallParams.m"}]],
```

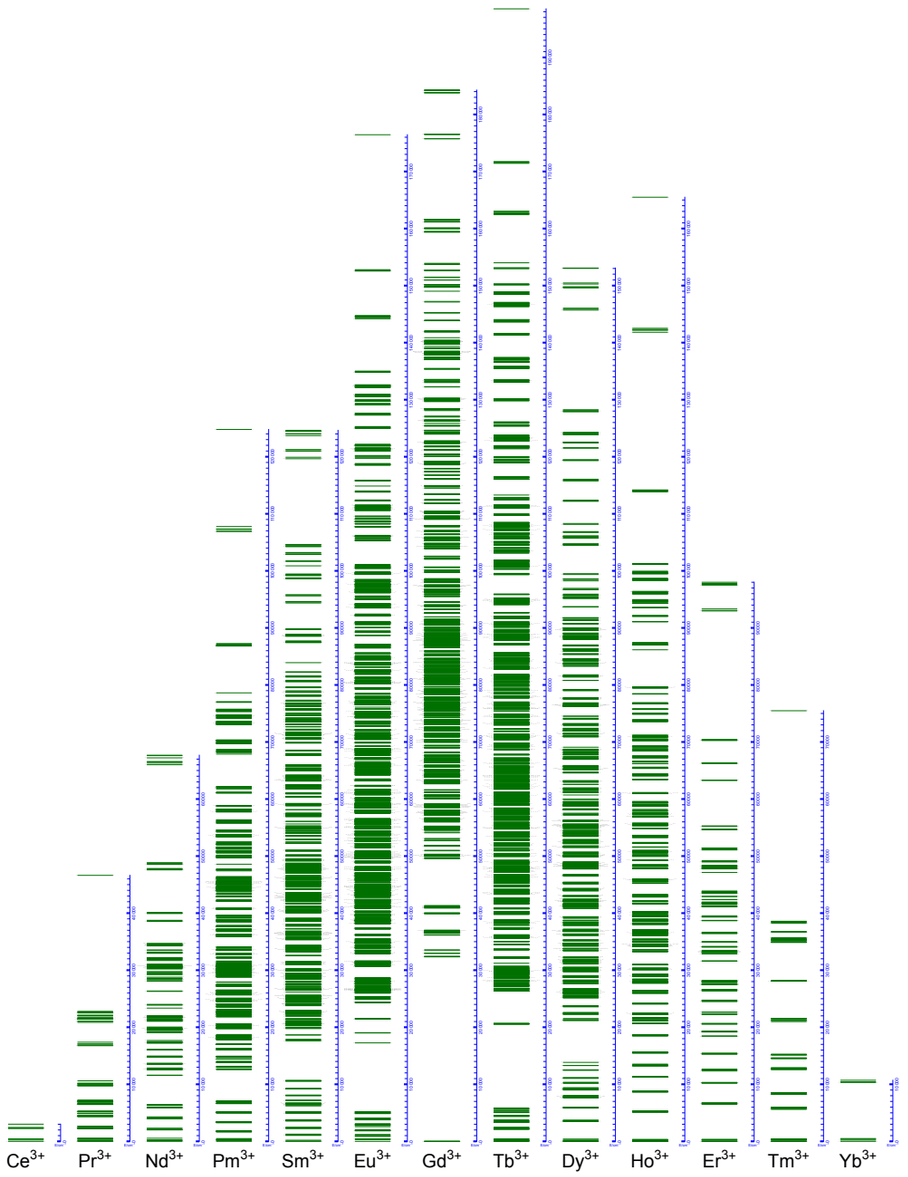


Figure 7: Energy levels in $LiYF_4$.

```
      OptionValue["Vintage"] == "Standard",
      Import[FileNameJoin[{moduleDir, "data", "standardParams.m"}]]
    ];
  params = If[OptionValue["With Uncertainties"],
    paramData[Ln],
    If[Head[#] === Around, #["Value"], #] & /@ paramData[Ln]
  ];
  Return[params];
  )
];
```

```
LoadLiYF4Parameters::usage="LoadLiYF4Parameters[ln] takes a string
    with the symbol the element of a trivalent lanthanide ion and
    returns model parameters for it. It return the data for LiYF4 from
     Cheng et al.";
LoadLiYF4Parameters[ln_, OptionsPattern[]]:=(
  If[!ValueQ[paramsLiYF4],
    paramsChengLiYF4 = Import[FileNameJoin[{moduleDir,"data","
    chengLiYF4.m"}]];
  ];
  Return[paramsChengLiYF4[ln]];
)
```

11 sparsefn.py

qlanth is also accompanied by seven Python scripts `sparsef[1-7].py`. Each of these contains a function `effective_hamiltonian_f[1-7]` which returns a sparse array (using this data structure as provided by `scipy`) for given values for the model parameters.

There is an eight Python script called `basisLSJMJ.py` which contains a dictionary whose keys are f1, f2, f3, f4, f5, f6, and f7, and whose values are lists that contain the ordered basis in which the array produced by the `sparsefn.py` should be understood to be in. Each basis vector is a list with three elements {LS string in NK notation, J, M_J}.

In those it is left up to the user to make the adequate change of signs in the parameters for configurations above $\underline{f}^7$. These include changing the signs of all in &qn-18 and setting `t2Switch` to 0. For configurations at or below $\underline{f}^7$ it is necessary to set `t2Switch` to 1.

12 Data sources

The data (and their provenance) upon which **qlanth** bases its calculations is the following:

- Coefficients of fractional parentage and seniority numbers from Velkov [Vel00].
- Terms labels from $\underline{f}^1$ to $\underline{f}^7$ from Nielson and Koster [NK63].
- 3j-symbol [Wol24b] and 6j-symbol [Wol24a] values from *Mathematica* (v 13.2),
- Reduced matrix elements for the three body operators from Judd [JS84].
- Reduced matrix elements for the magnetic interactions from Judd [JCC68].

13 Other details

- Fitting the experimental data for the entire row might take about 45 minutes, if run for the first time, but takes much less time once compiled functions from the truncated (or not truncated) Hamiltonian have been saved to disk.
- The code was run in *Mathematica* version 14.1 on MacOS Sequoia 15.2.

14 Units

Following the tradition of the spectroscopic community, all the matrix elements of the Hamiltonian are calculated using the Kayser ($\mathcal{K} \equiv \mathrm{cm}^{-1}$) as the (pseudo) energy unit. All the parameters (except the magnetic field which is in Tesla) in the effective Hamiltonian are assumed to be in this unit. As is customary, the angular momentum operators assume atomic units in which $\hbar = 1$.

Some constants and conversion values are included in the file `qonstants.m`.

```
BeginPackage["qonstants`"];

(* Spectroscopic niceties*)
theLanthanides = {"Ce", "Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy
    ", "Ho", "Er", "Tm", "Yb"};
theActinides   = {"Ac", "Th", "Pa", "U", "Np", "Pu", "Am", "Cm", "Bk"
    , "Cf", "Es", "Fm", "Md", "No", "Lr"};
theTrivalents  = {"Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho
    ", "Er", "Tm"};
specAlphabet   = "SPDFGHIKLMNOQRTUV";
complementaryNumE = {1,13,2,12,3,11,4,10,5,9,6,8,7};

(* SI *)
hPlanck = 6.62607015 * 10^-34;          (* Planck's constant in J s
    *)
hBar    = hPlanck    / (2 \[Pi]);       (* reduced Planck's constant
    in J s *)
\[Mu]B  = 9.2740100783 * 10^-24;        (* Bohr magneton in SI *)
me      = 9.1093837015 * 10^-31;        (* electron mass in kg *)
cLight  = 2.99792458   * 10^8;          (* speed of light in m/s *)
eCharge = 1.602176634  * 10^-19;        (* elementary charge in SI *)
\[Epsilon]0  = 8.8541878128 * 10^-12;   (* electric permittivity in
    vacuum in SI *)
```

```
\[Mu]0  = 4 \[Pi] * 10^-7;              (* magnetic permeability in
    vacuum in SI *)
αFine = 1/137.036;                (* fine structure constant *)
debye   = 1 / cLight * 10^-21;          (* debye in C m *)

bohrRadius    = 5.29177*10^-11;          (* Bohr radius in m *)
hartreeEnergy = hBar^2 / (me * bohrRadius^2); (* Hartree energy in J
    *)
hartreeTime   = hBar / hartreeEnergy; (* Hartree time in s *)

(* Hartree atomic units *)
hPlanckHartree = 2 \[Pi]; (* Planck's constant in Hartree *)
meHartree      = 1;       (* electron mass in Hartree *)
cLightHartree  = 137.036; (* speed of light in Hartree *)
eChargeHartree = 1;       (* elementary charge in Hartree *)
\[Mu]0Hartree  = αFine^2; (* magnetic permeability in vacuum in
    Hartree *)

(* some conversion factors *)
eVToJoule       = eCharge;
jouleToeV       = 1 / eVToJoule;
jouleToHartree  = 1 / hartreeEnergy;
eVToKayser      = eCharge /( hPlanck * cLight * 100 ); (* 1 eV =
    8065.54429 cm^-1 *)
kayserToeV      = 1 / eVToKayser;
teslaToKayser   = 2 * \[Mu]B / hPlanck / cLight / 100;
kayserToHartree = kayserToeV * eVToJoule * jouleToHartree;
hartreeToKayser = 1 / kayserToHartree;

EndPackage[];
```

15 Notation

orbital angular momentum operator of a single electron

$$\overbrace{\hat{l}} \tag{125}$$

total orbital angular momentum operator

$$\overbrace{\hat{L}} \tag{126}$$

spin angular momentum operator of a single electron

$$\overbrace{\hat{s}} \tag{127}$$

total spin angular momentum operator

$$\overbrace{\hat{S}} \tag{128}$$

Shorthand for all other quantum numbers

$$\overbrace{\Lambda} \tag{129}$$

orbital angular momentum number

$$\overbrace{\underline{\ell}} \tag{130}$$

spinning angular momentum number

$$\overbrace{\underline{s}} \tag{131}$$

Coulomb non-central potential

$$\overbrace{\hat{\mathfrak{C}}} \tag{132}$$

LS-reduced matrix element of operator $\hat{O}$ between ΛLS and $\Lambda' L' S'$

$$\overbrace{\langle \Lambda LS \| \hat{O} \| \Lambda' L' S' \rangle} \tag{133}$$

LSJ-reduced matrix element of operator $\hat{O}$ between ΛLSJ and $\Lambda' L' S' J'$

$$\overbrace{\langle \Lambda LSJ \| \hat{O} \| \Lambda' L' S' J' \rangle} \tag{134}$$

Spectroscopic term αLS in Russel-Saunders notation

$$\overbrace{{}^{2S+1}\alpha \mathrm{L} \equiv |\alpha LS\rangle} \tag{135}$$

spherical tensor operator of rank k

$$\overbrace{\hat{X}^{(k)}} \tag{136}$$

q-component of the spherical tensor operator $\hat{X}^{(k)}$

$$\overbrace{\hat{X}^{(k)}_q} \tag{137}$$

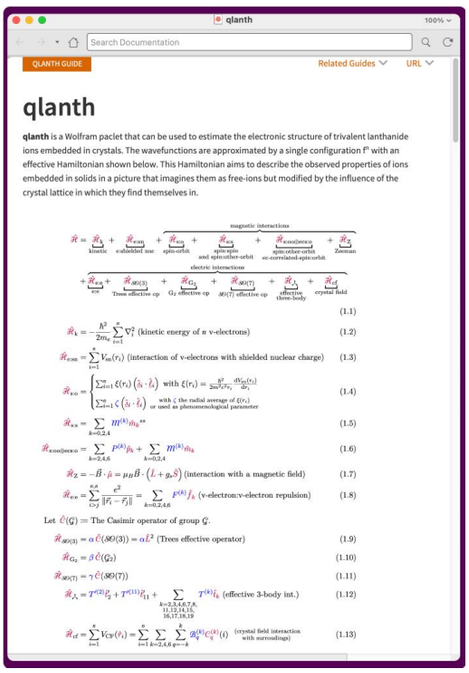
The coefficient of fractional parentage from the parent term $|\underline{\ell}^{n-1} \alpha' L' S'\rangle$ for the daughter term $|\underline{\ell}^n \alpha LS\rangle$

$$\overbrace{\left(\underline{\ell}^{n-1} \alpha' L' S' \middle\} \underline{\ell}^n \alpha LS\right)} \tag{138}$$

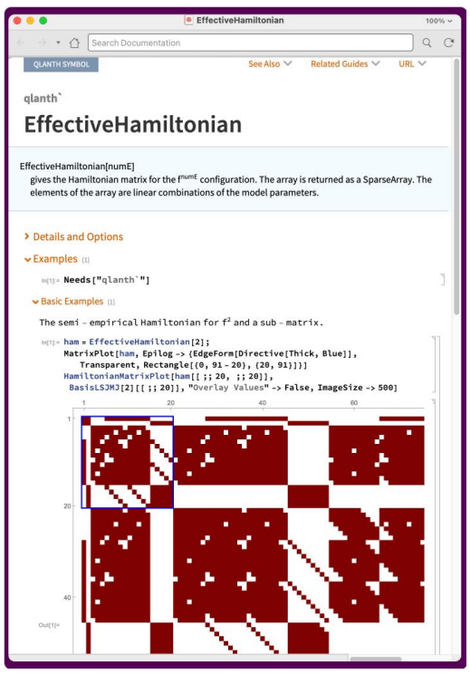
16 Mathematica `paclet`

The simplest way of adding **`qlanth`** to *Mathematica* is by installing the corresponding paclet. To do this, first download the `paclet` file for the latest release of the code on GitHub. Install the `paclet` file using PacletInstall within a *Mathematica* session. After this **`qlanth`** may be loaded into a notebook by issuing the command `Needs["qlanth`"]`.

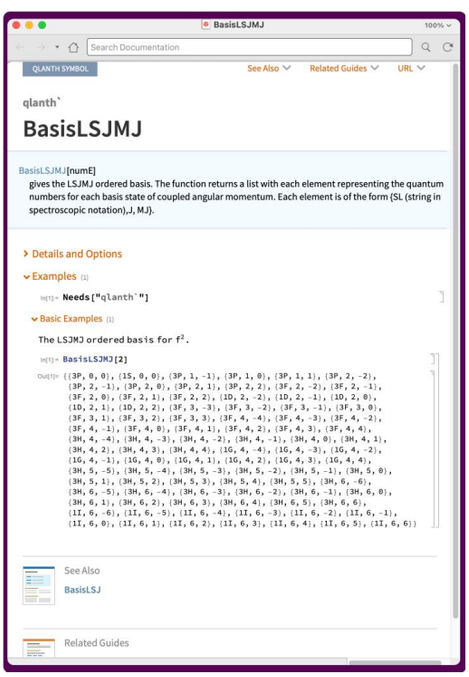
The paclet includes documentation in standard *Mathematica* format (see Fig-8a).



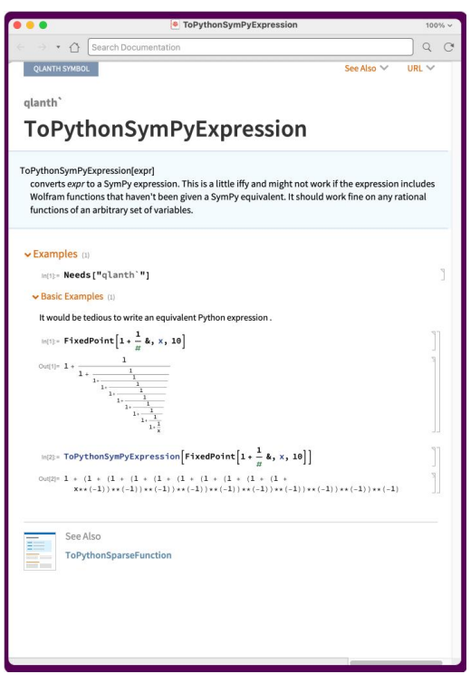
(a) **qlanth** guide in *Mathematica*



(b) Documentation for `EffectiveHamiltonian`



(c) Doc. for `BasisLSJMJ`



(d) Doc. for `ToPythonSymPyExpression`

17 Definitions

$$\overbrace{[\![x]\!]}^{\text{two plus one}} \coloneqq 2x+1 \tag{139}$$

$$\overbrace{\hat{u}^{(k)}}^{\text{irreducible unit tensor operator of rank k}} \tag{140}$$

$$\overbrace{\hat{U}^{(k)} \coloneqq \sum_{i=1}^{n} \hat{u}^{(k)}}^{\text{symmetric unit tensor operator for n equivalent electrons}} \tag{141}$$

$$\overbrace{C_q^{(k)} \coloneqq \sqrt{\frac{4\pi}{2k+1}} \mathrm{Y}_q^{(k)}}^{\text{Renormalized spherical harmonics}} \tag{142}$$

$$\overbrace{\triangleleft(j_1,j_2,j_3) \coloneqq \begin{cases} 1 & \text{if } j_1 = (j_2+j_3),(j_2+j_3-1),\ldots,|j_2-j_3| \\ 0 & \text{otherwise} \end{cases}}^{\text{Triangle “delta” between } j_1,j_2,j_3} \tag{143}$$

18 code

18.1 qlanth.m

This file encapsulates the main functions in **qlanth** and contains all the physics related functions.

```
(* ------------------------------------------------------------------
+--------------------------------------------------------------------+
|                                                                    |
|                                                                    |
|                    __                          __     __          |
|             ____ _   / /  ____ _   ____      / /_   / /_         |
|            / __ `/  / /  / __ `/  / __ \    / __/  / __ \        |
|           / /_/ /  / /  / /_/ /  / / / /  / /_   / / / /         |
|           \__, /  /_/   \__, _/  /_/ /_/   \__/  /_/ /_/         |
|             /_/                                                    |
|                                                                    |
|                                                                    |
+--------------------------------------------------------------------+
This code was initially authored by Christopher M. Dodson and Rashid
Zia, and then rewritten and expanded by Juan David Lizarazo Ferro in
the years 2022-2024 under the advisory of Dr. Rashid Zia. It has
also benefited from the discussions with Tharnier Puel at the
University of Iowa.

It grew out of a collaboration sponsored by the NSF (NSF
DMR-1922025) between the groups of Dr. Rashid Zia at Brown
University, the Quantum Engineering Laboratory at the University of
Pennsylvania led by Dr. Lee Bassett, and the group of Dr. Michael
Flatté at the University of Iowa.

It uses an effective Hamiltonian to describe the electronic
structure of lanthanide ions in crystals. This effective Hamiltonian
includes terms representing the following interactions/relativistic
corrections: spin-orbit, electrostatic repulsion, spin-spin, crystal
field, and spin-other-orbit.

The Hilbert space used in this effective Hamiltonian is limited to
single f^n configurations. The inaccuracy of this single
configuration description is partially compensated by the inclusion
of configuration interaction terms as parametrized by the Casimir
operators of SO(3), G(2), and SO(7), and by three-body effective
operators ti.

The parameters included in this model are listed in the string
paramAtlas.

The notebook "qlanth.nb" contains a gallery with many of the
functions included in this module with some simple use cases.

The notebook "The Lanthanides in LaF3.nb" is an example in which the
results from this code are compared against the published results by
Carnall et. al for the energy levels of lanthinde ions in crystals
of lanthanum trifluoride.

VERSION: MARCH 2025

REFERENCES:

+ Condon, E U, and G Shortley. "The Theory of Atomic Spectra." 1935.

+ Racah, Giulio. "Theory of Complex Spectra. II." Physical Review
62, no. 9-10 (November 1, 1942): 438-62.
https://doi.org/10.1103/PhysRev.62.438.

+ Racah, Giulio. "Theory of Complex Spectra. III." Physical Review
63, no. 9-10 (May 1, 1943): 367-82.
https://doi.org/10.1103/PhysRev.63.367.

+ Judd, B. R. "Optical Absorption Intensities of Rare-Earth Ions."
Physical Review 127, no. 3 (August 1, 1962): 750-61.
https://doi.org/10.1103/PhysRev.127.750.

+ Ofelt, GS. "Intensities of Crystal Spectra of Rare-Earth Ions."
The Journal of Chemical Physics 37, no. 3 (1962): 511-20.

```

```
+ Rajnak, K, and BG Wybourne. "Configuration Interaction Effects in l^N Configurations." Physical Review 132, no. 1 (1963): 280. https://doi.org/10.1103/PhysRev.132.280.

+ Nielson, C. W., and George F Koster. "Spectroscopic Coefficients for the p^n, d^n, and f^n Configurations", 1963.

+ Wybourne, Brian. "Spectroscopic Properties of Rare Earths." 1965.

+ Carnall, W To, PR Fields, and BG Wybourne. "Spectral Intensities of the Trivalent Lanthanides and Actinides in Solution. I. Pr3+, Nd3+, Er3+, Tm3+, and Yb3+." The Journal of Chemical Physics 42, no. 11 (1965): 3797-3806.

+ Judd, BR. "Three-Particle Operators for Equivalent Electrons." Physical Review 141, no. 1 (1966): 4. https://doi.org/10.1103/PhysRev.141.4.

+ Judd, BR, HM Crosswhite, and Hannah Crosswhite. "Intra-Atomic Magnetic Interactions for f Electrons." Physical Review 169, no. 1 (1968): 130. https://doi.org/10.1103/PhysRev.169.130.

+ (TASS) Cowan, Robert Duane. "The Theory of Atomic Structure and Spectra." Los Alamos Series in Basic and Applied Sciences 3. Berkeley: University of California Press, 1981.

+ Judd, BR, and MA Suskin. "Complete Set of Orthogonal Scalar Operators for the Configuration f^3." JOSA B 1, no. 2 (1984): 261-65. https://doi.org/10.1364/JOSAB.1.000261.

+ Carnall, W. T., G. L. Goodman, K. Rajnak, and R. S. Rana. "A Systematic Analysis of the Spectra of the Lanthanides Doped into Single Crystal LaF3." The Journal of Chemical Physics 90, no. 7 (1989): 3443-57. https://doi.org/10.1063/1.455853.

+ Thorne, Anne, Ulf Litzen, and Sveneric Johansson. "Spectrophysics: Principles and Applications." Springer Science & Business Media, 1999.

+ Hansen, JE, BR Judd, and Hannah Crosswhite. "Matrix Elements of Scalar Three-Electron Operators for the Atomic f-Shell." Atomic Data and Nuclear Data Tables 62, no. 1 (1996): 1-49. https://doi.org/10.1006/adnd.1996.0001.

+ Velkov, Dobromir. "Multi-Electron Coefficients of Fractional Parentage for the p, d, and f Shells." John Hopkins University, 2000. The B1F_ALL.TXT file is from this thesis.

+ Dodson, Christopher M., and Rashid Zia. "Magnetic Dipole and Electric Quadrupole Transitions in the Trivalent Lanthanide Series: Calculated Emission Rates and Oscillator Strengths." Physical Review B 86, no. 12 (September 5, 2012): 125102. https://doi.org/10.1103/PhysRevB.86.125102.

+ Hehlen, Markus P, Mikhail G Brik, and Karl W Kramer. "50th Anniversary of the Judd-Ofelt Theory: An Experimentalist's View of the Formalism and Its Application." Journal of Luminescence 136 (2013): 221-39.

+ Rudzikas, Zenonas. Theoretical Atomic Spectroscopy, 2007.

+ Benelli, Cristiano, and Dante Gatteschi. Introduction to Molecular Magnetism: From Transition Metals to Lanthanides. John Wiley & Sons, 2015.

+ Newman, DJ, and G Balasubramanian. "Parametrization of Rare-Earth Ion Transition Intensities." Journal of Physics C: Solid State Physics 8, no. 1 (1975): 37.

------------------------------------------------------------------ *)

BeginPackage["qlanth`"];
Get[FileNameJoin[{DirectoryName[$InputFileName], "qonstants.m"}]]
Get[FileNameJoin[{DirectoryName[$InputFileName], "misc.m"}]]

paramAtlas = "
```

```
E0: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
E1: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
E2: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
E3: linear combination of F_k, see eqn. (2-80) in Wybourne 1965

ζ: spin-orbit strength parameter.

F0: Direct Slater integral F^0, produces an overall shift of all
    energy levels.
F2: Direct Slater integral F^2
F4: Direct Slater integral F^4, possibly constrained by ratio to F^2
F6: Direct Slater integral F^6, possibly constrained by ratio to F^2

M0: 0th Marvin integral
M2: 2nd Marvin integral
M4: 4th Marvin integral
\[Sigma]SS: spin-spin override, if 0 spin-spin is omitted, if 1 then
    spin-spin is included

T2:  three-body effective operator parameter T^2  (non-orthogonal)
T2p: three-body effective operator parameter T^2' (orthogonalized T2)
T3:  three-body effective operator parameter T^3
T4:  three-body effective operator parameter T^4
T6:  three-body effective operator parameter T^6
T7:  three-body effective operator parameter T^7
T8:  three-body effective operator parameter T^8

T11p: three-body effective operator parameter T^11' (orthogonalized
    T11)
T12:  three-body effective operator parameter T^12
T14:  three-body effective operator parameter T^14
T15:  three-body effective operator parameter T^15
T16:  three-body effective operator parameter T^16
T17:  three-body effective operator parameter T^17
T18:  three-body effective operator parameter T^18
T19:  three-body effective operator parameter T^19

P0: pseudo-magnetic parameter P^0
P2: pseudo-magnetic parameter P^2
P4: pseudo-magnetic parameter P^4
P6: pseudo-magnetic parameter P^6

gs: electronic gyromagnetic ratio

α: Trees' parameter α describing configuration interaction via the
    Casimir operator of SO(3)
β: Trees' parameter β describing configuration interaction via the
    Casimir operator of G(2)
γ: Trees' parameter γ describing configuration interaction via the
    Casimir operator of SO(7)

B02: crystal field parameter B_0^2 (real)
B04: crystal field parameter B_0^4 (real)
B06: crystal field parameter B_0^6 (real)
B12: crystal field parameter B_1^2 (real)
B14: crystal field parameter B_1^4 (real)

B16: crystal field parameter B_1^6 (real)
B22: crystal field parameter B_2^2 (real)
B24: crystal field parameter B_2^4 (real)
B26: crystal field parameter B_2^6 (real)
B34: crystal field parameter B_3^4 (real)

B36: crystal field parameter B_3^6 (real)
B44: crystal field parameter B_4^4 (real)
B46: crystal field parameter B_4^6 (real)
B56: crystal field parameter B_5^6 (real)
B66: crystal field parameter B_6^6 (real)

S12: crystal field parameter S_1^2 (real)
S14: crystal field parameter S_1^4 (real)
S16: crystal field parameter S_1^6 (real)
S22: crystal field parameter S_2^2 (real)

S24: crystal field parameter S_2^4 (real)
S26: crystal field parameter S_2^6 (real)
```

```
S34: crystal field parameter S_3^4 (real)
S36: crystal field parameter S_3^6 (real)

S44: crystal field parameter S_4^4 (real)
S46: crystal field parameter S_4^6 (real)
S56: crystal field parameter S_5^6 (real)
S66: crystal field parameter S_6^6 (real)

\[Epsilon]: ground level baseline shift
t2Switch: controls the usage of the t2 operator beyond f7 (1 for f7
    or below, 0 for f8 or above)
wChErrA: If 1 then the type-A errors in Chen are used, if 0 then not.
wChErrB: If 1 then the type-B errors in Chen are used, if 0 then not.

Bx: x component of external magnetic field (in T)
By: y component of external magnetic field (in T)
Bz: z component of external magnetic field (in T)

\[CapitalOmega]2: Judd-Ofelt intensity parameter k=2 (in cm^2)
\[CapitalOmega]4: Judd-Ofelt intensity parameter k=4 (in cm^2)
\[CapitalOmega]6: Judd-Ofelt intensity parameter k=6 (in cm^2)

nE: number of electrons in a configuration

E0p: orthogonalized E0
E1p: orthogonalized E1
E2p: orthogonalized E2
E3p: orthogonalized E3
αp: orthogonalized α
βp: orthogonalized β
γp: orthogonalized γ
";
paramSymbols    = StringSplit[paramAtlas, "\n"];
paramSymbols    = Select[paramSymbols, # != ""& ];
paramSymbols    = ToExpression[StringSplit[#, ":"][[1]]] & /@
    paramSymbols;
Protect /@ paramSymbols;

(* Parameter families *)
Unprotect[racahSymbols, chenSymbols, slaterSymbols, controlSymbols,
    cfSymbols, TSymbols, pseudoMagneticSymbols, marvinSymbols,
    casimirSymbols, magneticSymbols, juddOfeltIntensitySymbols,
    mostlyOrthogonalSymbols];
racahSymbols    = {E0, E1, E2, E3};
chenSymbols     = {wChErrA, wChErrB};
slaterSymbols   = {F0, F2, F4, F6};
controlSymbols  = {t2Switch, \[Sigma]SS};
cfSymbols       = {B02, B04, B06, B12, B14, B16, B22, B24, B26, B34,
    B36,
                   B44, B46, B56, B66,
                   S12, S14, S16, S22, S24, S26, S34, S36, S44, S46,
    S56, S66};
TSymbols        = {T2, T2p, T3, T4, T6, T7, T8, T11p, T12, T14, T15,
    T16, T17, T18, T19};
pseudoMagneticSymbols = {P0, P2, P4, P6};
marvinSymbols         = {M0, M2, M4};
magneticSymbols       = {Bx, By, Bz, gs, ζ};
casimirSymbols        = {α, β, γ};
juddOfeltIntensitySymbols = {\[CapitalOmega]2, \[CapitalOmega]4, \[
    CapitalOmega]6};
mostlyOrthogonalSymbols = Join[{E0p, E1p, E2p, E3p, αp, βp, γp},
  {T2p, T3, T4, T6, T7, T8, T11p, T12, T14, T15, T16, T17, T18, T19},
  cfSymbols];

paramFamilies         = Hold[{racahSymbols, chenSymbols,
    slaterSymbols, controlSymbols, cfSymbols, TSymbols,
    pseudoMagneticSymbols, marvinSymbols, casimirSymbols,
    magneticSymbols, juddOfeltIntensitySymbols,
    mostlyOrthogonalSymbols}];
ReleaseHold[Protect /@ paramFamilies];
crystalGroups = {"C1","Ci","C2","Cs","C2h","D2","C2v","D2h","C4","S4"
    ,"C4h","D4","C4v","D3d","D4h","C3","C3i","D3","C3v","D3d","C6","
    C3h","C6h","D6","C6v","D3h","D6h","T","Th","O","Td","Oh"};

(* Parameter usage *)
paramLines = Select[StringSplit[paramAtlas, "\n"], # != "" &];
```

```
usageTemplate = StringTemplate["`paramSymbol`::usage=\"`paramSymbol`: `paramUsage`\";"];
Do[(
  {paramString, paramUsage} = StringSplit[paramLine, ":"];
  paramUsage                = StringTrim[paramUsage];
  expressionString          = usageTemplate[<|"paramSymbol" -> paramString, "paramUsage" -> paramUsage|>];
  ToExpression[usageTemplate[<|"paramSymbol" -> paramString, "paramUsage" -> paramUsage|>]]
),
{paramLine, paramLines}
];

AllowedJ;
AllowedMforJ;
AllowedNKSLJMforJMTerms;
AllowedNKSLJMforJTerms;
AllowedNKSLJTerms;

AllowedNKSLTerms;
AllowedNKSLforJTerms;
AllowedSLJMTerms;
AllowedSLJTerms;
AllowedSLTerms;

AngularMomentumMatrices;
BasisLSJ;
BasisLSJMJ;
Bqk;
CFP;

CFPAssoc;
CFPTable;
CFPTerms;
Carnall;
CasimirG2;

CasimirSO3;
CasimirSO7;
Ck;
Cqk;
CrystalField;
CrystalFieldForm;

Dk;
EigenLever;
EffectiveHamiltonian;
EffectiveElectricDipole;
Electrostatic;
ElectrostaticConfigInteraction;

ElectrostaticTable;
EnergyLevelDiagram;
EnergyStates;
EtoF;
ExportMZip;

FindNKLSTerm;
FindSL;
FreeHam;
FreeIonTable;
ElectricDipLineStrength;
FromArrayToTable;
FtoE;

GG2U;
GSO7W;
GenerateCFP;
GenerateCFPAssoc;
GenerateCFPTable;

GenerateCrystalFieldTable;
GenerateElectrostaticTable;
GenerateFreeIonTable;
GenerateReducedUkTable;
GenerateReducedV1kTable;
```

```
GenerateSOOandECSOLSTable;

GenerateSOOandECSOTable;
GenerateSpinOrbitTable;
GenerateSpinSpinTable;
GenerateT22Table;
GenerateThreeBodyTables;

GroundMagDipoleOscillatorStrength;
HamiltonianForm;

HamiltonianMatrixPlot;
HoleElectronConjugation;
ImportMZip;
IonSolver;
JJBlockMagDip;

JJBlockMatrix;
JJBlockMatrixFileName;
JJBlockMatrixTable;
JuddCFPPhase;
JuddOfeltUkSquared;
LabeledGrid;
LandeFactor;

LevelElecDipoleOscillatorStrength;
LevelJJBlockMagDipole;
LevelMagDipoleLineStrength;
LevelMagDipoleMatrixAssembly;
LevelMagDipoleOscillatorStrength;

LevelMagDipoleSpontaneousDecayRates;
LevelSimplerEffectiveHamiltonian;
LevelSolver;
LoadAll;

LoadCFP;
LoadCarnall;
LoadChenDeltas;
LoadElectrostatic;
LoadFreeIon;

LoadLaF3Parameters;
LoadLiYF4Parameters;
LoadSOOandECSO;
LoadSOOandECSOLS;
LoadSpinOrbit;
LoadSpinSpin;

LoadT22;
LoadTermLabels;
LoadThreeBody;

LoadUk;
LoadV1k;
MagDipLineStrength;
MagDipoleMatrixAssembly;
MagDipoleRates;

MagneticInteractions;
MapToSparseArray;
MaxJ;
MinJ;
NKCFPPhase;

ParamPad;
ParseBenelli2015;
ParseStates;
ParseStatesByNumBasisVecs;
ParseStatesByProbabilitySum;

ParseTermLabels;
Phaser;
PrettySaundersSL;
PrettySaundersSLJ;
PrettySaundersSLJmJ;
```

```
426
427 PrintL;
428 PrintSLJ;
429 PrintSLJM;
430 ReducedSOOandECSOinf2;
431 ReducedSOOandECSOinfn;
432
433 ReducedT11inf2;
434 ReducedT22inf2;
435 ReducedT22infn;
436 ReducedUk;
437 ReducedUkTable;
438
439 ReducedV1k;
440 ReducedV1kTable;
441 Reducedt11inf2;
442 ReplaceInSparseArray;
443 ScalarLSJMFromLS;
444 SOOandECSO;
445 SOOandECSOLSTable;
446
447 SOOandECSOTable;
448 Seniority;
449 ShiftedLevels;
450 SimplerEffectiveHamiltonian;
451
452 SixJay;
453 SpinOrbit;
454 SpinOrbitTable;
455 SpinSpin;
456 SpinSpinTable;
457
458 Sqk;
459 SquarePrimeToNormal;
460 TPO;
461 T22Table;
462 TabulateJJBlockMagDipTable;
463 TabulateJJBlockMatrixTable;
464
465 TabulateManyJJBlockMagDipTables;
466 TabulateManyJJBlockMatrixTables;
467 ThreeBodyTable;
468 ThreeBodyTables;
469 ThreeJay;
470 UkOperator;
471
472 chenDeltas;
473 fnTermLabels;
474 fsubk;
475
476 fsupk;
477 moduleDir;
478 symbolicHamiltonians;
479
480 (* this selects the function that is applied to calculated matrix
      elements which helps keep down the complexity of the resulting
      algebraic expressions *)
481 SimplifyFun = Expand;
482
483 Begin["`Private`"]
484
485   ListRepeater;
486   TotalCFIters;
487
488   moduleDir = ParentDirectory[DirectoryName[$InputFileName]];
489   frontEndAvailable = (Head[$FrontEnd] === FrontEndObject);
490
491   (* ############################################################ *)
492   (* ########################## MISC ############################ *)
493
494   TPO::usage = "TPO[x, y, ...] gives the product of 2x+1, 2y+1, ...";
495   TPO[args__] := Times @@ ((2*# + 1) & /@ {args});
496
497   Phaser::usage = "Phaser[x] gives (-1)^x.";
498   Phaser[exponent_] := ((-1)^exponent);
499
```

```
TriangleCondition::usage = "TriangleCondition[a, b, c] evaluates the triangle condition on a, b, and c.";
TriangleCondition[a_, b_, c_] := (Abs[b - c] <= a <= (b + c));

TriangleAndSumCondition::usage = "TriangleAndSumCondition[a, b, c] evaluates the joint satisfaction of the triangle and sum conditions.";
TriangleAndSumCondition[a_, b_, c_] := (
  And[
    Abs[b - c] <= a <= (b + c),
    IntegerQ[a + b + c]
    ]
  );

SquarePrimeToNormal::usage = "SquarePrimeToNormal[squarePrime] evaluates the standard representation of a number from the squared prime representation given in the list squarePrime. For squarePrime of the form {c0, c1, c2, c3, ...} this function returns the number c0 * Sqrt[p1^c1 * p2^c2 * p3^c3 * ...] where pi is the ith prime number. Exceptionally some of the ci might be letters in which case they have to be one of \"A\", \"B\", \"C\", \"D\" with them corresponding to 10, 11, 12, and 13, respectively.";
SquarePrimeToNormal[squarePrime_] :=
(
  radical = Product[Prime[idx1 - 1] ^ Part[squarePrime, idx1], {idx1, 2, Length[squarePrime]}];
  radical = radical /. {"A" -> 10, "B" -> 11, "C" -> 12, "D" -> 13};
  val = squarePrime[[1]] * Sqrt[radical];
  Return[val];
);

ParamPad::usage = "ParamPad[params] takes an association params whose keys are a subset of paramSymbols. The function returns a new association where all the keys not present in paramSymbols, will now be included in the returned association with their values set to zero.
The function additionally takes an option \"Print\" that if set to True, will print the symbols that were not present in the given association. The default is True.";
Options[ParamPad] = {"PrintFun" -> PrintTemporary};
ParamPad[params_, OptionsPattern[]] := (
  notPresentSymbols = Complement[paramSymbols, Keys[params]];
  PrintFun = OptionValue["PrintFun"];
  PrintFun["Following symbols were not given and are being set to 0: ",
    notPresentSymbols];
  newParams = Transpose[{paramSymbols, ConstantArray[0, Length[paramSymbols]]}];
  newParams = (#[[1]] -> #[[2]]) & /@ newParams;
  newParams = Association[newParams];
  newParams = Join[newParams, params];
  Return[newParams];
  )

(* ############################################################ *)
(* ##################### Angular Momentum ##################### *)

AngularMomentumMatrices::usage = "AngularMomentumMatrices[j] gives the matrix representation for the angular momentum operators Jx, Jy, and Jz for a given angular momentum j in the basis of eigenvectors of jz. j may be a half-integer or an integer.
The options are \"Sparse\" which defaults to False and \"Order\" which defaults to \"HighToLow\".
The option \"Order\" can be set to \"LowToHigh\" to get the matrices in the order from -jay to jay otherwise they are returned in the order jay to -jay.
The function returns a list {JxMatrix, JyMatrix, JzMatrix} with the matrix representations for the cartesian components of the angular momentum operator.";
Options[AngularMomentumMatrices] = {
  "Sparse" -> False,
  "Order" -> "HighToLow"};
AngularMomentumMatrices[jay_, OptionsPattern[]] := Module[
  {
```

```
547       JxMatrix, JyMatrix, JzMatrix,
548       JPlusMatrix, JMinusMatrix,
549       m1, m2,
550       ArrayInverter
551     },
552     (
553       ArrayInverter = #[[-1 ;; 1 ;; -1, -1 ;; 1 ;; -1]] &;
554       JPlusMatrix   = Table[
555         If[m2 == m1 + 1,
556           Sqrt[(jay - m1) (jay + m1 + 1)],
557           0
558           ],
559         {m1, jay, -jay, -1},
560         {m2, jay, -jay, -1}];
561       JMinusMatrix = Table[
562         If[m2 == m1 - 1,
563           Sqrt[(jay + m1) (jay - m1 + 1)],
564           0
565           ],
566         {m1, jay, -jay, -1},
567         {m2, jay, -jay, -1}];
568       JxMatrix = (JPlusMatrix + JMinusMatrix)/2;
569       JyMatrix = (JMinusMatrix - JPlusMatrix)/(2 I);
570       JzMatrix = DiagonalMatrix[Table[m, {m, jay, -jay, -1}]];
571       If[OptionValue["Sparse"],
572         {JxMatrix, JyMatrix, JzMatrix} = SparseArray /@ {JxMatrix,
    JyMatrix, JzMatrix}
573       ];
574       If[OptionValue["Order"] == "LowToHigh",
575         {JxMatrix, JyMatrix, JzMatrix} = ArrayInverter /@ {JxMatrix,
    JyMatrix, JzMatrix};
576       ];
577       Return[{JxMatrix, JyMatrix, JzMatrix}];
578     )
579   ];
580
581   LandeFactor::usage="LandeFactor[J, L, S] gives the Lande factor for
       a given total angular momentum J, orbital angular momentum L, and
       spin S.";
582   LandeFactor[J_, L_, S_] := (3/2) + (S*(S + 1) - L*(L + 1))/(2*J*(J
    + 1));
583
584   (* ##################### Angular Momentum ##################### *)
585   (* ############################################################ *)
586
587
588   (* ############################################################ *)
589   (* ###################### Racah Algebra ###################### *)
590
591   ReducedUk::usage = "ReducedUk[n, l, SL, SpLp, k] gives the reduced
      matrix element of the symmetric unit tensor operator U^(k). See
      equation 11.53 in TASS.";
592   ReducedUk[numE_, l_, SL_, SpLp_, k_] := Module[
593     {spin, orbital, Uk, S, L,
594     Sp, Lp, Sb, Lb, parentSL,
595     cfpSL, cfpSpLp, Ukval,
596     SLparents, SLpparents,
597     commonParents, phase},
598       {spin, orbital} = {1/2, 3};
599       {S, L}          = FindSL[SL];
600       {Sp, Lp}        = FindSL[SpLp];
601       If[Not[S == Sp],
602         Return[0]
603       ];
604       cfpSL      = CFP[{numE, SL}];
605       cfpSpLp    = CFP[{numE, SpLp}];
606       SLparents  = First /@ Rest[cfpSL];
607       SLpparents = First /@ Rest[cfpSpLp];
608       commonParents = Intersection[SLparents, SLpparents];
609       Uk = Sum[(
610         {Sb, Lb} = FindSL[\[Psi]b];
611         Phaser[Lb] *
612           CFPAssoc[{numE, SL, \[Psi]b}] *
613           CFPAssoc[{numE, SpLp, \[Psi]b}] *
614           SixJay[{orbital, k, orbital}, {L, Lb, Lp}]
615       ),
```

```
      {\[Psi]b, commonParents}
      ];
      phase     = Phaser[orbital + L + k];
      prefactor = numE * phase * Sqrt[TPO[L,Lp]];
      Ukval     = prefactor*Uk;
      Return[Ukval];
  ]

  Ck::usage = "Ck[orbital, k] gives the diagonal reduced matrix
    element <l||C^(k)||l> where the Subscript[C, q]^(k) are
    renormalized spherical harmonics. See equation 11.23 in TASS with
    l=l'.";
  Ck[orbital_, k_] := (-1)^orbital * TPO[orbital] * ThreeJay[{orbital
    , 0}, {k, 0}, {orbital, 0}];

  SixJay::usage = "SixJay[{j1, j2, j3}, {j4, j5, j6}] provides the
    value for SixJSymbol[{j1, j2, j3}, {j4, j5, j6}] with memorization
     of computed values and short-circuiting values based on triangle
    conditions.";
  SixJay[{j1_, j2_, j3_}, {j4_, j5_, j6_}] := (
    sixJayval = Which[
      Not[TriangleAndSumCondition[j1, j2, j3]],
      0,
      Not[TriangleAndSumCondition[j1, j5, j6]],
      0,
      Not[TriangleAndSumCondition[j4, j2, j6]],
      0,
      Not[TriangleAndSumCondition[j4, j5, j3]],
      0,
      True,
      SixJSymbol[{j1, j2, j3}, {j4, j5, j6}]];
    SixJay[{j1, j2, j3}, {j4, j5, j6}] = sixJayval);

  ThreeJay::usage = "ThreeJay[{j1, m1}, {j2, m2}, {j3, m3}] gives the
     value of the Wigner 3j-symbol and memorizes the computed value.";
  ThreeJay[{j1_, m1_}, {j2_, m2_}, {j3_, m3_}] := (
   threejval = Which[
     Not[(m1 + m2 + m3) == 0],
     0,
     Not[TriangleCondition[j1,j2,j3]],
     0,
     True,
     ThreeJSymbol[{j1, m1}, {j2, m2}, {j3, m3}]
     ];
   ThreeJay[{j1, m1}, {j2, m2}, {j3, m3}] = threejval);

  ReducedV1k::usage = "ReducedV1k[n, SL, SpLp, k] gives the reduced
    matrix element of the spherical tensor operator V^(1k). See
    equation 2-101 in Wybourne 1965.";
  ReducedV1k[numE_, SL_, SpLp_, k_] := Module[
    {Vk1, S, L, Sp, Lp,
    Sb, Lb, spin, orbital,
    cfpSL, cfpSpLp,
    SLparents, SpLpparents,
    commonParents, prefactor},
    (
      {spin, orbital} = {1/2, 3};
      {S, L}          = FindSL[SL];
      {Sp, Lp}        = FindSL[SpLp];
      cfpSL           = CFP[{numE, SL}];
      cfpSpLp         = CFP[{numE, SpLp}];
      SLparents       = First /@ Rest[cfpSL];
      SpLpparents     = First /@ Rest[cfpSpLp];
      commonParents = Intersection[SLparents, SpLpparents];
      Vk1 = Sum[(
          {Sb, Lb} = FindSL[\[Psi]b];
          Phaser[(Sb + Lb + S + L + orbital + k - spin)] *
          CFPAssoc[{numE, SL, \[Psi]b}] *
          CFPAssoc[{numE, SpLp, \[Psi]b}] *
          SixJay[{S, Sp, 1}, {spin, spin, Sb}] *
          SixJay[{L, Lp, k}, {orbital, orbital, Lb}]
        ),
      {\[Psi]b, commonParents}
      ];
      prefactor = numE * Sqrt[spin * (spin + 1) * TPO[spin, S, L, Sp,
    Lp] ];
```

```
681       Return[prefactor * Vk1];
682     )
683   ];
684
685   GenerateReducedUkTable::usage = "GenerateReducedUkTable[numEmax]
        can be used to generate the association of reduced matrix elements
        for the unit tensor operators Uk from f^1 up to f^numEmax. If the
        option \"Export\" is set to True then the resulting data is saved
        to ./data/ReducedUkTable.m.";
686   Options[GenerateReducedUkTable] = {"Export" -> True, "Progress" ->
        True};
687   GenerateReducedUkTable[numEmax_Integer:7, OptionsPattern[]] := (
688     numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
        AllowedNKSLTerms[#]]&/@Range[1, numEmax]] * 4;
689     Echo["Calculating " <> ToString[numValues] <> " values for Uk k
        =0,2,4,6."];
690     counter = 1;
691     If[And[OptionValue["Progress"], frontEndAvailable],
692     progBar = PrintTemporary[
693         Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
694           counter}]]]
695       ];
696     ReducedUkTable = Table[
697       (
698         counter = counter+1;
699         {numE, 3, SL, SpLp, k} -> SimplifyFun[ReducedUk[numE, 3, SL,
        SpLp, k]]
700       ),
701       {numE, 1, numEmax},
702       {SL,   AllowedNKSLTerms[numE]},
703       {SpLp, AllowedNKSLTerms[numE]},
704       {k, {0, 2, 4, 6}}
705     ];
706     ReducedUkTable = Association[Flatten[ReducedUkTable]];
707     ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
        ReducedUkTable.m"}];
708     If[And[OptionValue["Progress"], frontEndAvailable],
709       NotebookDelete[progBar]
710     ];
711     If[OptionValue["Export"],
712       (
713         Echo["Exporting to file " <> ToString[ReducedUkTableFname]];
714         Export[ReducedUkTableFname, ReducedUkTable];
715       )
716     ];
717     Return[ReducedUkTable];
718   )
719
720   GenerateReducedV1kTable::usage = "GenerateReducedV1kTable[nmax]
        calculates values for Vk1 and returns an association where the
        keys are lists of the form {n, SL, SpLp, 1}. If the option \"
        Export\" is set to True then the resulting data is saved to ./data
        /ReducedV1kTable.m.";
721   Options[GenerateReducedV1kTable] = {"Export" -> True, "Progress" ->
         True};
722   GenerateReducedV1kTable[numEmax_Integer:7, OptionsPattern[]] := (
723     numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
        AllowedNKSLTerms[#]]&/@Range[1, numEmax]];
724     Echo["Calculating " <> ToString[numValues] <> " values for Vk1."
        ];
725     counter = 1;
726     If[And[OptionValue["Progress"], frontEndAvailable],
727     progBar = PrintTemporary[
728         Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
729           counter}]]]
730       ];
731     ReducedV1kTable = Table[
732       (
733         counter = counter+1;
734         {n, SL, SpLp, 1} -> SimplifyFun[ReducedV1k[n, SL, SpLp, 1]]
735       ),
736       {n, 1, numEmax},
737       {SL, AllowedNKSLTerms[n]},
738       {SpLp, AllowedNKSLTerms[n]}
739     ];
740     ReducedV1kTable = Association[ReducedV1kTable];
```

```
741     If[And[OptionValue["Progress"], frontEndAvailable],
742       NotebookDelete[progBar]
743     ];
744     exportFname = FileNameJoin[{moduleDir, "data", "ReducedV1kTable.m
        "}];
745     If[OptionValue["Export"],
746       (
747         Echo["Exporting to file " <> ToString[exportFname]];
748         Export[exportFname, ReducedV1kTable];
749       )
750     ];
751     Return[ReducedV1kTable];
752   )
753
754   (* ######################## Racah Algebra ######################## *)
755   (* ############################################################## *)
756
757   (* ############################################################## *)
758   (* ######################## Electrostatic ######################## *)
759
760   fsubk::usage = "fsubk[numE, orbital, SL, SLp, k] gives the Slater
        integral f_k for the given configuration and pair of SL terms. See
         equation 12.17 in TASS.";
761   fsubk[numE_, orbital_, NKSL_, NKSLp_, k_] := Module[
762     {terms, S, L, Sp, Lp,
763     termsWithSameSpin, SL,
764     fsubkVal, spinMultiplicity,
765     prefactor, summand1, summand2},
766     (
767       {S, L}   = FindSL[NKSL];
768       {Sp, Lp} = FindSL[NKSLp];
769       terms    = AllowedNKSLTerms[numE];
770       (* sum for summand1 is over terms with same spin *)
771       spinMultiplicity  = 2*S + 1;
772       termsWithSameSpin = StringCases[terms, ToString[
        spinMultiplicity] ~~ __];
773       termsWithSameSpin = Flatten[termsWithSameSpin];
774       If[Not[{S, L} == {Sp, Lp}],
775         Return[0]
776       ];
777       prefactor = 1/2 * Abs[Ck[orbital, k]]^2;
778       summand1 = Sum[(
779           ReducedUkTable[{numE, orbital, SL, NKSL,  k}] *
780           ReducedUkTable[{numE, orbital, SL, NKSLp, k}]
781           ),
782         {SL, termsWithSameSpin}
783       ];
784       summand1 = 1 / TPO[L] * summand1;
785       summand2 = (
786         KroneckerDelta[NKSL, NKSLp] *
787           (numE *(4*orbital + 2 - numE)) /
788           ((2*orbital + 1) * (4*orbital + 1))
789         );
790       fsubkVal = prefactor*(summand1 - summand2);
791       Return[fsubkVal];
792     )
793   ];
794
795   fsupk::usage = "fsupk[numE, orbital, SL, SLp, k] gives the
        superscripted Slater integral f^k = Subscript[f, k] * Subscript[D,
         k].";
796   fsupk[numE_, orbital_, NKSL_, NKSLp_ ,k_] := (
797     Dk[k] * fsubk[numE, orbital, NKSL, NKSLp, k]
798     )
799
800   Dk::usage = "D[k] gives the ratio between the super-script and sub-
        scripted Slater integrals (F^k / F_k). k must be even. See table
        6-3 in TASS, and also section 2-7 of Wybourne (1965). See also
        equation 6.41 in TASS.";
801   Dk[k_] := {1, 225, 1089, 184041/25}[[k/2+1]];
802
803   FtoE::usage = "FtoE[F0, F2, F4, F6] calculates the Racah parameters
         {E0, E1, E2, E3} corresponding to the given Slater integrals.
804   See eqn. 2-80 in Wybourne.
805   Note that in that equation the subscripted Slater integrals are
        used but since this function assumes the the input values are
```

```
  superscripted Slater integrals, it is necessary to convert them
  using Dk.";
FtoE[F0_, F2_, F4_, F6_] := Module[
  {E0, E1, E2, E3},
  (
    E0 = (F0 - 10*F2/Dk[2] - 33*F4/Dk[4] - 286*F6/Dk[6]);
    E1 = (70*F2/Dk[2] + 231*F4/Dk[4] + 2002*F6/Dk[6])/9;
    E2 = (F2/Dk[2] - 3*F4/Dk[4] + 7*F6/Dk[6])/9;
    E3 = (5*F2/Dk[2] + 6*F4/Dk[4] - 91*F6/Dk[6])/3;
    Return[{E0, E1, E2, E3}];
  )
];

EtoF::usage = "EtoF[E0, E1, E2, E3] calculates the Slater integral
  parameters {F0, F2, F4, F6} corresponding to the given Racah
  parameters {E0, E1, E2, E3}. This is the inverse of the FtoE
  function.";
EtoF[E0_, E1_, E2_, E3_] := Module[
  {F0, F2, F4, F6},
  (
    F0 = 1/7      (7 E0 + 9 E1);
    F2 = 75/14    (E1 + 143 E2 + 11 E3);
    F4 = 99/7     (E1 - 130 E2 + 4 E3);
    F6 = 5577/350 (E1 + 35 E2 - 7 E3);
    Return[{F0, F2, F4, F6}];
  )
];

Electrostatic::usage = "Electrostatic[{numE, NKSL, NKSLp}] returns
  the LS reduced matrix element for repulsion matrix element for
  equivalent electrons. See equation 2-79 in Wybourne (1965). The
  option \"Coefficients\" can be set to \"Slater\" or \"Racah\". If
  set to \"Racah\" then E_k parameters and e^k operators are assumed
  , otherwise the Slater integrals F^k and operators f_k. The
  default is \"Slater\".";
Options[Electrostatic] = {"Coefficients" -> "Slater"};
Electrostatic[{numE_, NKSL_, NKSLp_}, OptionsPattern[]] := Module[
  {fsub0, fsub2, fsub4, fsub6,
   esub0, esub1, esub2, esub3,
   fsup0, fsup2, fsup4, fsup6,
   eMatrixVal, orbital},
  (
    orbital = 3;
    Which[
      OptionValue["Coefficients"] == "Slater",
      (
        fsub0 = fsubk[numE, orbital, NKSL, NKSLp, 0];
        fsub2 = fsubk[numE, orbital, NKSL, NKSLp, 2];
        fsub4 = fsubk[numE, orbital, NKSL, NKSLp, 4];
        fsub6 = fsubk[numE, orbital, NKSL, NKSLp, 6];
        eMatrixVal = fsub0*F0 + fsub2*F2 + fsub4*F4 + fsub6*F6;
      ),
      OptionValue["Coefficients"] == "Racah",
      (
        fsup0 = fsupk[numE, orbital, NKSL, NKSLp, 0];
        fsup2 = fsupk[numE, orbital, NKSL, NKSLp, 2];
        fsup4 = fsupk[numE, orbital, NKSL, NKSLp, 4];
        fsup6 = fsupk[numE, orbital, NKSL, NKSLp, 6];
        esub0 = fsup0;
        esub1 = 9/7*fsup0 +   1/42*fsup2 +   1/77*fsup4 +  1/462*
  fsup6;
        esub2 =             143/42*fsup2 - 130/77*fsup4 + 35/462*
  fsup6;
        esub3 =              11/42*fsup2 + 4/77*fsup4   -  7/462*
  fsup6;
        eMatrixVal = esub0*E0 + esub1*E1 + esub2*E2 + esub3*E3;
      )
    ];
    Return[eMatrixVal];
  )
];

GenerateElectrostaticTable::usage = "GenerateElectrostaticTable[
  numEmax] can be used to generate the table for the electrostatic
  interaction from f^1 to f^numEmax. If the option \"Export\" is set
   to True then the resulting data is saved to ./data/
```

```
  ElectrostaticTable.m.";
Options[GenerateElectrostaticTable] = {"Export" -> True, "
  Coefficients" -> "Slater"};
GenerateElectrostaticTable[numEmax_Integer:7, OptionsPattern[]] :=
  (
   ElectrostaticTable = Table[
     {numE, SL, SpLp} -> SimplifyFun[Electrostatic[{numE, SL, SpLp},
    "Coefficients" -> OptionValue["Coefficients"]]],
     {numE, 1, numEmax},
     {SL, AllowedNKSLTerms[numE]},
     {SpLp, AllowedNKSLTerms[numE]}
   ];
   ElectrostaticTable = Association[Flatten[ElectrostaticTable]];
   If[OptionValue["Export"],
     Export[FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"
  }],
     ElectrostaticTable];
   ];
   Return[ElectrostaticTable];
);

(* ###################### Electrostatic ###################### *)
(* ########################################################### *)

(* ########################################################### *)
(* ########################### Bases ########################## *)

BasisTableGenerator::usage = "BasisTableGenerator[numE] gives an
  association whose keys are lists of the form {numE, J} and whose
  values are lists with elements of the form {LS, J, MJ}
  representing the elements of the LSJM coupled basis.";
BasisTableGenerator[numE_] := Module[
  {energyStatesTable, allowedJ, J, Jp},
  (
    energyStatesTable = <||>;
    allowedJ = AllowedJ[numE];
    Do[
    (
      energyStatesTable[{numE, J}] = EnergyStates[numE, J];
    ),
    {Jp, allowedJ},
    {J,  allowedJ}];
    Return[energyStatesTable]
  )
];

BasisLSJMJ::usage = "BasisLSJMJ[numE] returns the ordered basis in
  L-S-J-MJ with the total orbital angular momentum L and total spin
  angular momentum S coupled together to form J. The function
  returns a list with each element representing the quantum numbers
  for each basis vector. Each element is of the form {SL (string in
  spectroscopic notation),J, MJ}.
The option \"AsAssociation\" can be set to True to return the basis
   as an association with the keys corresponding to values of J and
  the values lists with the corresponding {L, S, J, MJ} list. The
  default of this option is False.
";
Options[BasisLSJMJ] = {"AsAssociation" -> False};
BasisLSJMJ[numE_, OptionsPattern[]] := Module[
  {energyStatesTable, basis, idx1},
  (
    energyStatesTable = BasisTableGenerator[numE];
    basis = Table[
      energyStatesTable[{numE, AllowedJ[numE][[idx1]]}],
      {idx1, 1, Length[AllowedJ[numE]]}];
    basis = Flatten[basis, 1];
    If[OptionValue["AsAssociation"],
      (
        Js    = AllowedJ[numE];
        basis = Table[(J -> Select[basis, #[[2]] == J &]), {J, Js
  }];
        basis = Association[basis];
      )
    ];
    Return[basis];
  )
```

```
];

BasisLSJ::usage = "BasisLSJ[numE] returns the level basis LSJ. The
  function returns a list with each element representing the quantum
   numbers for each basis vector. Each element is of the form {SL (
  string in spectroscopic notation), J}.
The option \"AsAssociation\" can be set to True to return the basis
   as an association with the keys being the allowed J values. The
  default is False.
  ";
Options[BasisLSJ]={"AsAssociation"->False};
BasisLSJ[numE_,OptionsPattern[]] := Module[
  {Js, basis},
  (
    Js    = AllowedJ[numE];
    basis = BasisLSJMJ[numE,"AsAssociation"->False];
    basis = DeleteDuplicates[{#[[1]],#[[2]]}& /@ basis];
    If[OptionValue["AsAssociation"],
      (
        basis = Association @ Table[(J->Select[basis,#[[2]]==J&]),{
  J,Js}]
      )
    ];
    Return[basis];
  )
];

(* ########################## Bases ########################### *)
(* ############################################################ *)

(* ############################################################ *)
(* ########### Coefficients of Fracional Parentage ########### *)

GenerateCFP::usage = "GenerateCFP[] generates the association for
  the coefficients of fractional parentage. Result is exported to
  the file ./data/CFP.m. The coefficients of fractional parentage
  are taken beyond the half-filled shell using the phase convention
  determined by the option \"PhaseFunction\". The default is \"NK\"
  which corresponds to the phase convention of Nielson and Koster.
  The other option is \"Judd\" which corresponds to the phase
  convention of Judd.";
Options[GenerateCFP] = {"Export" -> True, "PhaseFunction"-> "NK"};
GenerateCFP[OptionsPattern[]] := (
  CFP = Table[
    {numE, NKSL} -> First[CFPTerms[numE, NKSL]],
    {numE, 1, 7},
    {NKSL, AllowedNKSLTerms[numE]}];
  CFP = Association[CFP];
  (* Go all the way to f14 *)
  CFP = CFPExpander["Export" -> False, "PhaseFunction"->
  OptionValue["PhaseFunction"]];
  If[OptionValue["Export"],
    Export[FileNameJoin[{moduleDir, "data", "CFPs.m"}], CFP];
  ];
  Return[CFP];
);

JuddCFPPhase::usage = "Phase between conjugate coefficients of
  fractional parentage according to Velkov's thesis, page 40.";
JuddCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
  parentSeniority_, daughterSeniority_] := Module[
  {spin, orbital, expo, phase},
  (
    {spin, orbital} = {1/2, 3};
    expo = (
        (parentS + parentL + daughterS + daughterL) -
        (orbital + spin) +
        1/2 * (parentSeniority + daughterSeniority - 1)
    );
    phase = Phaser[-expo];
    Return[phase];
  )
];

NKCFPPhase::usage = "Phase between conjugate coefficients of
  fractional parentage according to Nielson and Koster page viii.
```

```
  Note that there is a typo on there the expression for zeta should
  be (-1)^((v-1)/2) instead of (-1)^(v - 1/2).";
NKCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
  parentSeniority_, daughterSeniority_] := Module[
  {spin, orbital, expo, phase},
  (
    {spin, orbital} = {1/2, 3};
    expo = (
        (parentS + parentL + daughterS + daughterL) -
        (orbital + spin)
    );
    phase = Phaser[-expo];
    If[parent == 2*orbital,
        phase = phase * Phaser[(daughterSeniority-1)/2]];
    Return[phase];
  )
];

Options[CFPExpander] = {"Export" -> True, "PhaseFunction" -> "NK"};
CFPExpander::usage = "Using the coefficients of fractional
  parentage up to f7 this function calculates them up to f14.
The coefficients of fractional parentage are taken beyond the half-
  filled shell using the phase convention determined by the option \
  "PhaseFunction\". The default is \"NK\" which corresponds to the
  phase convention of Nielson and Koster. The other option is \"Judd
  \" which corresponds to the phase convention of Judd. The result
  is exported to the file ./data/CFPs_extended.m.";
CFPExpander[OptionsPattern[]] := Module[
  {orbital, halfFilled, fullShell, parentMax, PhaseFun,
  complementaryCFPs, daughter, conjugateDaughter,
  conjugateParent, parentTerms, daughterTerms,
  parentCFPs, daughterSeniority, daughterS, daughterL,
  parentCFP, parentTerm, parentCFPval,
  parentS, parentL, parentSeniority, phase, prefactor,
  newCFPval, key, extendedCFPs, exportFname},
  (
    orbital    = 3;
    halfFilled = 2 * orbital + 1;
    fullShell  = 2 * halfFilled;
    parentMax  = 2 * orbital;

    PhaseFun  = <|
        "Judd" -> JuddCFPPhase,
        "NK" -> NKCFPPhase|>[OptionValue["PhaseFunction"]];
    PrintTemporary["Calculating CFPs using the phase system from ",
   PhaseFun];
    (* Initialize everything with lists to be filled in the next Do
  *)
    complementaryCFPs =
        Table[
        ({numE, term} -> {term}),
        {numE, halfFilled + 1, fullShell - 1, 1},
        {term, AllowedNKSLTerms[numE]
        }];
    complementaryCFPs = Association[Flatten[complementaryCFPs]];
    Do[(
        daughter          = parent + 1;
        conjugateDaughter = fullShell - parent;
        conjugateParent   = conjugateDaughter - 1;
        parentTerms       = AllowedNKSLTerms[parent];
        daughterTerms     = AllowedNKSLTerms[daughter];
        Do[
        (
            parentCFPs             = Rest[CFP[{daughter,
  daughterTerm}]];
            daughterSeniority      = Seniority[daughterTerm];
            {daughterS, daughterL} = FindSL[daughterTerm];
            Do[
            (
                {parentTerm, parentCFPval} = parentCFP;
                {parentS, parentL}         = FindSL[parentTerm];
                parentSeniority            = Seniority[parentTerm];
                phase = PhaseFun[parent, parentS, parentL,
                                 daughterS, daughterL,
                                 parentSeniority, daughterSeniority
  ];
```

```
                    prefactor = (daughter * TPO[daughterS, daughterL])
    /
                                (conjugateDaughter * TPO[parentS,
    parentL]);
                    prefactor = Sqrt[prefactor];
                    newCFPval = phase * prefactor * parentCFPval;
                    key = {conjugateDaughter, parentTerm};
                    complementaryCFPs[key] = Append[complementaryCFPs[
    key], {daughterTerm, newCFPval}]
                ),
                {parentCFP, parentCFPs}
                ]
            ),
            {daughterTerm, daughterTerms}
            ]
            ),
        {parent, 1, parentMax}
        ];

        complementaryCFPs[{14, "1S"}] = {"1S", {"2F",1}};
        extendedCFPs          = Join[CFP, complementaryCFPs];
        If[OptionValue["Export"];,
        (
            exportFname = FileNameJoin[{moduleDir, "data", "
    CFPs_extended.m"}];
            Echo["Exporting to " <> exportFname];
            Export[exportFname, extendedCFPs];
        )
        ];
        Return[extendedCFPs];
    )
    ];

    GenerateCFPTable::usage = "GenerateCFPTable[] generates the table
      for the coefficients of fractional parentage. If the optional
      parameter \"Export\" is set to True then the resulting data is
      saved to ./data/CFPTable.m.
    The data being parsed here is the file attachment B1F_ALL.TXT which
       comes from Velkov's thesis.";
    Options[GenerateCFPTable] = {"Export" -> True};
    GenerateCFPTable[OptionsPattern[]] := Module[
      {rawText, rawLines, leadChar, configIndex, line, daughter,
      lineParts, numberCode, parsedNumber, toAppend, CFPTablefname},
      (
        CleanWhitespace[string_]     := StringReplace[string,
      RegularExpression["\\s+"]->" "];
        AddSpaceBeforeMinus[string_] := StringReplace[string,
      RegularExpression["(?<!\\s)-"]->" -"];
        ToIntegerOrString[list_]     := Map[If[StringMatchQ[#,
      NumberString], ToExpression[#], #] &, list];
        CFPTable      = ConstantArray[{},7];
        CFPTable[[1]] = {{"2F",{"1S",1}}};

        (* Cleaning before processing is useful *)
        rawText  = Import[FileNameJoin[{moduleDir, "data", "B1F_ALL.TXT
    "}]];
        rawLines = StringTrim/@StringSplit[rawText,"\n"];
        rawLines = Select[rawLines,#!=""&];
        rawLines = CleanWhitespace/@rawLines;
        rawLines = AddSpaceBeforeMinus/@rawLines;

        Do[(
          (* the first character can be used to identify the start of a
       block *)
          leadChar=StringTake[line,{1}];
          (* ..FN, N is at position 50 in that line *)
          If[leadChar=="[",
          (
            configIndex=ToExpression[StringTake[line,{50}]];
            Continue[];
          )
          ];
          (* Identify which daughter term is being listed *)
          If[StringContainsQ[line,"[DAUGHTER TERM]"],
            daughter=StringSplit[line,"["][[1]];
            CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{
```

```
    daughter}];
1109          Continue[];
1110        ];
1111        (* Once we get here we are already parsing a row with
    coefficient data *)
1112        lineParts    = StringSplit[line," "];
1113        parent       = lineParts[[1]];
1114        numberCode   = ToIntegerOrString[lineParts[[3;;]]];
1115        parsedNumber = SquarePrimeToNormal[numberCode];
1116        toAppend     = {parent,parsedNumber};
1117        CFPTable[[configIndex]][[-1]] = Append[CFPTable[[configIndex
    ]][[-1]], toAppend]
1118      ),
1119      {line,rawLines}];
1120      If[OptionValue["Export"],
1121        (
1122        CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"
    }];
1123        Export[CFPTablefname, CFPTable];
1124        )
1125      ];
1126      Return[CFPTable];
1127    )
1128  ];
1129
1130  GenerateCFPAssoc::usage = "GenerateCFPAssoc[] converts the
      coefficients of fractional parentage into an association in which
      zero values are explicit. If the option \"Export\" is set to True,
       the association is exported to the file /data/CFPAssoc.m. This
      function requires that the association CFP be defined.";
1131  Options[GenerateCFPAssoc] = {"Export" -> True};
1132  GenerateCFPAssoc[OptionsPattern[]] := (
1133    CFPAssoc = Association[];
1134    Do[
1135      (daughterTerms = AllowedNKSLTerms[numE];
1136      parentTerms    = AllowedNKSLTerms[numE - 1];
1137      Do[
1138        (
1139        cfps = CFP[{numE, daughter}];
1140        cfps = cfps[[2 ;;]];
1141        parents = First /@ cfps;
1142        Do[
1143          (
1144          key = {numE, daughter, parent};
1145          cfp = If[
1146            MemberQ[parents, parent],
1147            (
1148              idx = Position[parents, parent][[1, 1]];
1149              cfps[[idx]][[2]]
1150            ),
1151            0
1152            ];
1153          CFPAssoc[key] = cfp;
1154          ),
1155          {parent, parentTerms}
1156          ]
1157        ),
1158        {daughter, daughterTerms}
1159        ]
1160      ),
1161      {numE, 1, 14}
1162      ];
1163    If[OptionValue["Export"],
1164      (
1165      CFPAssocfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"
    }];
1166      Export[CFPAssocfname, CFPAssoc];
1167      )
1168    ];
1169    Return[CFPAssoc];
1170  );
1171
1172  CFPTerms::usage = "CFPTerms[numE] gives all the daughter and parent
      terms, together with the corresponding coefficients of fractional
      parentage, that correspond to the the f^n configuration.
1173  CFPTerms[numE, SL] gives all the daughter and parent terms,
```

```
    together with the corresponding coefficients of fractional
    parentage, that are compatible with the given string SL in the f^n
     configuration.
CFPTerms[numE, L, S] gives all the daughter and parent terms,
    together with the corresponding coefficients of fractional
    parentage, that correspond to the given total orbital angular
    momentum L and total spin S n the f^n configuration. L being an
    integer, and S being integer or half-integer.
In all cases the output is in the shape of a list with enclosed
    lists having the format {daughter_term, {parent_term_1, CFP_1}, {
    parent_term_2, CFP_2}, ...}.
Only the one-body coefficients for f-electrons are provided.
In all cases it must be that 1 <= n <= 7.
These are according to the tables from Nielson & Koster.
";
CFPTerms[numE_] := Part[CFPTable, numE]
CFPTerms[numE_, SL_] := Module[
  {NKterms, CFPconfig},
  (
    NKterms   = {{}};
    CFPconfig = CFPTable[[numE]];
    Map[
      If[StringFreeQ[First[#], SL],
        Null,
        NKterms = Join[NKterms, {#}, 1]
      ] &,
    CFPconfig
    ];
    NKterms = DeleteCases[NKterms, {}]
  )
];
CFPTerms[numE_, L_, S_] := Module[
{NKterms, SL, CFPconfig},
(
  SL = StringJoin[ToString[2 S + 1], PrintL[L]];
  NKterms = {{}};
  CFPconfig = Part[CFPTable, numE];
  Map[
    If[StringFreeQ[First[#], SL],
      Null,
      NKterms = Join[NKterms, {#}, 1]
    ]&,
  CFPconfig
  ];
  NKterms = DeleteCases[NKterms, {}]
)
];

(* ########### Coefficients of Fracional Parentage ########### *)
(* ############################################################ *)

(* ############################################################ *)
(* ######################## Spin Orbit ######################## *)

SpinOrbit::usage = "SpinOrbit[numE, SL, SpLp, J] returns the LSJ
    reduced matrix element ζ <SL, J|L.S|SpLp, J>. These are given as a
     function of ζ. This function requires that the association
    ReducedV1kTable be defined.
See equations 2-106 and 2-109 in Wybourne (1965). Equivalently see
    eqn. 12.43 in TASS.";
SpinOrbit[numE_, SL_, SpLp_, J_] := Module[
  {S, L, Sp, Lp, orbital, sign, prefactor, val},
  (
    orbital   = 3;
    {S, L}    = FindSL[SL];
    {Sp, Lp}  = FindSL[SpLp];
    prefactor = Sqrt[orbital * (orbital+1) * (2*orbital+1)] *
                SixJay[{L, Lp, 1}, {Sp, S, J}];
    sign      = Phaser[J + L + Sp];
    val       = sign * prefactor * ζ * ReducedV1kTable[{numE, SL,
  SpLp, 1}];
    Return[val];
  )
];

SpinOrbitTable::usage="An association containing the matrix
```

```
    elements for the spin-orbit interaction for f^n configurations.
    The keys are lists of the form {n, SL, SpLp, J}.";
1236
1237  GenerateSpinOrbitTable::usage = "GenerateSpinOrbitTable[nmax]
    computes the matrix elements for the spin-orbit interaction for f^
    n configurations up to n = nmax. The function returns an
    association whose keys are lists of the form {n, SL, SpLp, J}. If
    \"Export\" is set to True, then the result is exported to the data
     folder. It requires ReducedV1kTable to be defined.";
1238  Options[GenerateSpinOrbitTable] = {"Export" -> True};
1239  GenerateSpinOrbitTable[nmax_Integer:7, OptionsPattern[]] := Module[
1240    {numE, J, SL, SpLp, exportFname},
1241    (
1242      SpinOrbitTable =
1243        Table[
1244          {numE, SL, SpLp, J} -> SpinOrbit[numE, SL, SpLp, J],
1245        {numE, 1, nmax},
1246        {J, MinJ[numE], MaxJ[numE]},
1247        {SL,   Map[First, AllowedNKSLforJTerms[numE, J]]},
1248        {SpLp, Map[First, AllowedNKSLforJTerms[numE, J]]}
1249        ];
1250      SpinOrbitTable = Association[SpinOrbitTable];
1251
1252      exportFname = FileNameJoin[{moduleDir, "data", "SpinOrbitTable.
    m"}];
1253      If[OptionValue["Export"],
1254        (
1255          Echo["Exporting to file " <> ToString[exportFname]];
1256          Export[exportFname, SpinOrbitTable];
1257        )
1258      ];
1259      Return[SpinOrbitTable];
1260    )
1261  ];
1262
1263  (* ********************** Spin Orbit ************************** *)
1264  (* ************************************************************ *)
1265
1266  (* ************************************************************ *)
1267  (* ****************** Three Body Operators ********************* *)
1268
1269  ParseJudd1984::usage = "This function parses the data from tables 1
     and 2 of Judd from Judd, BR, and MA Suskin. \"Complete Set of
    Orthogonal Scalar Operators for the Configuration f^3\". JOSA B 1,
     no. 2 (1984): 261-65.";
1270  Options[ParseJudd1984] = {"Export" -> False};
1271  ParseJudd1984[OptionsPattern[]] := (
1272    ParseJuddTab1[str_] := (
1273      strR = ToString[str];
1274      strR = StringReplace[strR, ".5" -> "^(1/2)"];
1275      num = ToExpression[strR];
1276      sign = Sign[num];
1277      num = sign*Simplify[Sqrt[num^2]];
1278      If[Round[num] == num, num = Round[num]];
1279      Return[num]);
1280
1281    (* Parse table 1 from Judd 1984 *)
1282    judd1984Fname1 = FileNameJoin[{moduleDir, "data", "Judd1984-1.csv
    "}];
1283    data = Import[judd1984Fname1, "CSV", "Numeric" -> False];
1284    headers = data[[1]];
1285    data = data[[2 ;;]];
1286    data = Transpose[data];
1287    \[Psi] = Select[data[[1]], # != "" &];
1288    \[Psi]p = Select[data[[2]], # != "" &];
1289    matrixKeys = Transpose[{\[Psi], \[Psi]p}];
1290    data = data[[3 ;;]];
1291    cols = Table[ParseJuddTab1 /@ Select[col, # != "" &], {col, data
    }];
1292    cols = Select[cols, Length[#] == 21 &];
1293    tab1 = Prepend[Prepend[cols, \[Psi]p], \[Psi]];
1294    tab1 = Transpose[Prepend[Transpose[tab1], headers]];
1295
1296    (* Parse table 2 from Judd 1984 *)
1297    judd1984Fname2 = FileNameJoin[{moduleDir, "data", "Judd1984-2.csv
    "}];
```

```
    data = Import[judd1984Fname2, "CSV", "Numeric" -> False];
    headers = data[[1]];
    data = data[[2 ;;]];
    data = Transpose[data];
    {operatorLabels, WUlabels, multiFactorSymbols, multiFactorValues}
     = data[[;; 4]];
    multiFactorValues = ParseJuddTab1 /@ multiFactorValues;
    multiFactorValues = AssociationThread[multiFactorSymbols ->
   multiFactorValues];

    (*scale values of table 1 given the values in table 2*)
    oppyS = {};
    normalTable =
      Table[header = col[[1]];
        If[StringContainsQ[header, " "],
          (
            multiplierSymbol = StringSplit[header, " "][[1]];
            multiplierValue = multiFactorValues[multiplierSymbol];
            operatorSymbol = StringSplit[header, " "][[2]];
            oppyS = Append[oppyS, operatorSymbol];
          ),
          (
            multiplierValue = 1;
            operatorSymbol = header;
          )
        ];
        normalValues = 1/multiplierValue*col[[2 ;;]];
        Join[{operatorSymbol}, normalValues], {col, tab1[[3 ;;]]}
      ];

    (*Create an association for the reduced matrix elements in the f
   ^3 config*)
    juddOperators = Association[];
    Do[(
      col      = normalTable[[colIndex]];
      opLabel  = col[[1]];
      opValues = col[[2 ;;]];
      opMatrix = AssociationThread[matrixKeys -> opValues];
      Do[(
        opMatrix[Reverse[mKey]] = opMatrix[mKey]
        ),
      {mKey, matrixKeys}
      ];
      juddOperators[{3, opLabel}] = opMatrix),
      {colIndex, 1, Length[normalTable]}
    ];

    (* special case of t2 in f3 *)
    (* this is the same as getting the reduced matrix elements from
   Judd 1966 *)
    numE = 3;
    e3Op      = juddOperators[{3, "e_{3}"}];
    t2prime   = juddOperators[{3, "t_{2}^{'}"}];
    prefactor = 1/(70 Sqrt[2]);
    t2Op = (# -> (t2prime[#] + prefactor*e3Op[#])) & /@ Keys[t2prime
   ];
    t2Op = Association[t2Op];
    juddOperators[{3, "t_{2}"}] = t2Op;

    (*Special case of t11 in f3*)
    t11 = juddOperators[{3, "t_{11}"}];
    eβprimeOp = juddOperators[{3, "e_{\\beta}^{'}"}];
    t11primeOp = (# -> (t11[#] + Sqrt[3/385] eβprimeOp[#])) & /@ Keys
   [t11];
    t11primeOp = Association[t11primeOp];
    juddOperators[{3, "t_{11}^{'}"}] = t11primeOp;
    If[OptionValue["Export"],
      (
        (*export them*)
        PrintTemporary["Exporting ..."];
        exportFname = FileNameJoin[{moduleDir, "data", "juddOperators
   .m"}];
        Export[exportFname, juddOperators];
      )
    ];
    Return[juddOperators];
```

```
);

ThreeBodyTable::usage="ThreeBodyTable is an association containing the LS-reduced matrix elements for the three-body operators for f^n configurations. The keys are lists of the form {n, SL, SpLp}.";

ThreeBodyTables::usage="ThreeBodyTables is an association whose keys are integers n from 1 to 7 and whose values are associations whose keys are symbols for the different three-body operators, and whose keys are of the form {LS, LpSp} where LS and LpSp are strings for LS-terms in f^n.";

GenerateThreeBodyTables::usage = "This function generates the reduced matrix elements for the three body operators using the coefficients of fractional parentage, including those beyond f^7.";
Options[GenerateThreeBodyTables] = {"Export" -> False};
GenerateThreeBodyTables[OptionsPattern[]] := (
  tiKeys        = (StringReplace[ToString[#], {"T" -> "t_{", "p" -> "}^{'"}] <> "}") & /@ TSymbols;
  TSymbolsAssoc = AssociationThread[tiKeys -> TSymbols];
  juddOperators = ParseJudd1984[];
  (* op3MatrixElement[SL, SpLp, opSymbol] returns the value for the reduced matrix element of the operator opSymbol for the terms {SL, SpLp} in the f^3 configuration. *)
  op3MatrixElement[SL_, SpLp_, opSymbol_] := (
    jOP = juddOperators[{3, opSymbol}];
    key = {SL, SpLp};
    val = If[MemberQ[Keys[jOP], key],
      jOP[key],
      0];
    Return[val];
  );
  (* ti: This is the implementation of formula (2) in Judd & Suskin 1984. It computes the reduced matrix elements of ti in f^n by using the reduced matrix elements in f^3 and the coefficients of fractional parentage. If the option \"Fast\" is set to True then the values for n>7 are simply computed as the negatives of the values in the complementary configuration; this except for t2 and t11 which are treated as special cases. *)
  Options[ti] = {"Fast" -> True};
  ti[nE_, SL_, SpLp_, tiKey_, opOrder_ : 3, OptionsPattern[]] :=
  Module[
    {nn, S, L, Sp, Lp,
    cfpSL, cfpSpLp,
    parentSL, parentSpLp,
    tnk, tnks},
    (
      {S, L}   = FindSL[SL];
      {Sp, Lp} = FindSL[SpLp];
      fast     = OptionValue["Fast"];
      numH     = 14 - nE;
      If[fast && Not[MemberQ[{"t_{2}","t_{11}"},tiKey]] && nE > 7,
        Return[-tktable[{numH, SL, SpLp, tiKey}]]
      ];
      If[(S == Sp && L == Lp),
      (
        cfpSL   = CFP[{nE, SL}];
        cfpSpLp = CFP[{nE, SpLp}];
        tnks = Table[(
            parentSL   = cfpSL[[nn, 1]];
            parentSpLp = cfpSpLp[[mm, 1]];
            cfpSL[[nn, 2]] * cfpSpLp[[mm, 2]] *
            tktable[{nE - 1, parentSL, parentSpLp, tiKey}]
            ),
            {nn, 2, Length[cfpSL]},
            {mm, 2, Length[cfpSpLp]}
            ];
        tnk = Total[Flatten[tnks]];
        ),
      tnk = 0;
      ];
      Return[ nE / (nE - opOrder) * tnk];
    )
  ];
  (* Calculate the reduced matrix elements of t^i for n up to 14 *)
```

```
    tktable = <||>;
    Do[(
      Do[(
        tkValue = Which[numE <= 2,
          (*Initialize n=1,2 with zeros*)
          0,
          numE == 3,
          (* Grab matrix elem in f^3 from Judd 1984 *)
          SimplifyFun[op3MatrixElement[SL, SpLp, opKey]],
          True,
          SimplifyFun[ti[numE, SL, SpLp, opKey, If[opKey == "e_{3}",
    2, 3]]]
          ];
        tktable[{numE, SL, SpLp, opKey}] = tkValue;
        ),
      {SL, AllowedNKSLTerms[numE]},
      {SpLp, AllowedNKSLTerms[numE]},
      {opKey, Append[tiKeys, "e_{3}"]}
      ];
      PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], "
    configuration complete"]];
      ),
      {numE, 1, 14}
    ];

    (* Now use those reduced matrix elements to determine their sum
    as weighted by their corresponding strengths Ti *)
    ThreeBodyTable = <||>;
    Do[
      Do[
      (
        ThreeBodyTable[{numE, SL, SpLp}] = (
          Sum[(
            If[tiKey == "t_{2}", t2Switch, 1] *
            tktable[{numE, SL, SpLp, tiKey}] *
            TSymbolsAssoc[tiKey] +
            If[tiKey == "t_{2}", 1 - t2Switch, 0] *
            (-tktable[{14 - numE, SL, SpLp, tiKey}]) *
            TSymbolsAssoc[tiKey]
            ),
          {tiKey, tiKeys}
          ]
        );
      ),
      {SL, AllowedNKSLTerms[numE]},
      {SpLp, AllowedNKSLTerms[numE]}
      ];
    PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], " matrix
    complete"]];,
    {numE, 1, 7}
    ];

    ThreeBodyTables = Table[(
      terms = AllowedNKSLTerms[numE];
      singleThreeBodyTable =
        Table[
          {SL, SLp} -> ThreeBodyTable[{numE, SL, SLp}],
          {SL, terms},
          {SLp, terms}
        ];
      singleThreeBodyTable  = Flatten[singleThreeBodyTable];
      singleThreeBodyTables = Table[(
          notNullPosition = Position[TSymbols, notNullSymbol][[1,
    1]];
          reps = ConstantArray[0, Length[TSymbols]];
          reps[[notNullPosition]] = 1;
          rep = AssociationThread[TSymbols -> reps];
          notNullSymbol -> Association[(singleThreeBodyTable /. rep)]
          ),
        {notNullSymbol, TSymbols}
        ];
      singleThreeBodyTables = Association[singleThreeBodyTables];
      numE -> singleThreeBodyTables),
      {numE, 1, 7}
  ];
```

```
1495  ThreeBodyTables = Association[ThreeBodyTables];
1496  If[OptionValue["Export"],
1497    (
1498      threeBodyTablefname = FileNameJoin[{moduleDir, "data", "
        ThreeBodyTable.m"}];
1499      Export[threeBodyTablefname, ThreeBodyTable];
1500      threeBodyTablesfname = FileNameJoin[{moduleDir, "data", "
        ThreeBodyTables.m"}];
1501      Export[threeBodyTablesfname, ThreeBodyTables];
1502    )
1503   ];
1504  Return[{ThreeBodyTable, ThreeBodyTables}];
1505  );
1506
1507
1508  (* ################## Three Body Operators ##################### *)
1509  (* ############################################################ *)
1510
1511  (* ############################################################ *)
1512  (* ################### Reduced SOO and ECSO ##################### *)
1513
1514  ReducedT11inf2::usage = "ReducedT11inf2[SL, SpLp] returns the
       reduced matrix element of the scalar component of the double
       tensor T11 for the given SL terms SL, SpLp.
1515  Data used here for m0, m2, m4 is from Table II of Judd, BR, HM
       Crosswhite, and Hannah Crosswhite. Intra-Atomic Magnetic
       Interactions for f Electrons. Physical Review 169, no. 1 (1968):
       130.
1516  ";
1517  ReducedT11inf2[SL_, SpLp_] := Module[
1518    {T11inf2},
1519    (
1520      T11inf2 = <|
1521        {"1S", "3P"} -> 6 M0 + 2 M2 + 10/11 M4,
1522        {"3P", "3P"} -> -36 M0 - 72 M2 - 900/11 M4,
1523        {"3P", "1D"} -> -Sqrt[(2/15)] (27 M0 + 14 M2 + 115/11 M4),
1524        {"1D", "3F"} -> Sqrt[2/5] (23 M0 + 6 M2 - 195/11 M4),
1525        {"3F", "3F"} -> 2 Sqrt[14] (-15 M0 - M2 + 10/11 M4),
1526        {"3F", "1G"} -> Sqrt[11] (-6 M0 + 64/33 M2 - 1240/363 M4),
1527        {"1G", "3H"} -> Sqrt[2/5] (39 M0 - 728/33 M2 - 3175/363 M4),
1528        {"3H", "3H"} -> 8/Sqrt[55] (-132 M0 + 23 M2 + 130/11 M4),
1529        {"3H", "1I"} -> Sqrt[26] (-5 M0 - 30/11 M2 - 375/1573 M4)
1530        |>;
1531      Which[
1532        MemberQ[Keys[T11inf2],{SL,SpLp}],
1533          Return[T11inf2[{SL,SpLp}]],
1534        MemberQ[Keys[T11inf2],{SpLp,SL}],
1535          Return[T11inf2[{SpLp,SL}]],
1536        True,
1537          Return[0]
1538        ]
1539    )
1540  ];
1541
1542  Reducedt11inf2::usage = "Reducedt11inf2[SL, SpLp] returns the
       reduced matrix element in f^2 of the double tensor operator t11
       for the corresponding given terms {SL, SpLp}.
1543  Values given here are those from Table VII of \"Judd, BR, HM
       Crosswhite, and Hannah Crosswhite. \"Intra-Atomic Magnetic
       Interactions for f Electrons.\" Physical Review 169, no. 1 (1968):
        130.\"
1544  ";
1545  Reducedt11inf2[SL_, SpLp_] := Module[
1546    {t11inf2},
1547    (
1548      t11inf2 = <|
1549        {"1S", "3P"} -> -2 P0 - 105 P2 - 231 P4 - 429 P6,
1550        {"3P", "3P"} -> -P0 - 45 P2 - 33 P4 + 1287 P6,
1551        {"3P", "1D"} -> Sqrt[15/2] (P0 + 32 P2 - 33 P4 - 286 P6),
1552        {"1D", "3F"} -> Sqrt[10] (-P0 - 9/2 P2 + 66 P4 - 429/2 P6),
1553        {"3F", "3F"} -> Sqrt[14] (-P0 + 10 P2 + 33 P4 + 286 P6),
1554        {"3F", "1G"} -> Sqrt[11] (P0 - 20 P2 + 32 P4 - 104 P6),
1555        {"1G", "3H"} -> Sqrt[10] (-P0 + 55/2 P2 - 23 P4 - 65/2 P6),
1556        {"3H", "3H"} -> Sqrt[55] (-P0 + 25 P2 + 51 P4 + 13 P6),
1557        {"3H", "1I"} -> Sqrt[13/2] (P0 - 21 P4 - 6 P6)
1558        |>;
```

```
      Which[
        MemberQ[Keys[t11inf2],{SL,SpLp}],
          Return[t11inf2[{SL,SpLp}]],
        MemberQ[Keys[t11inf2],{SpLp,SL}],
          Return[t11inf2[{SpLp,SL}]],
        True,
          Return[0]
      ]
    )
];

ReducedSOOandECSOinf2::usage = "ReducedSOOandECSOinf2[SL, SpLp]
  returns the reduced matrix element corresponding to the operator (
  T11 + t11 - a13 * z13 / 6) for the terms {SL, SpLp}. This
  combination of operators corresponds to the spin-other-orbit plus
  ECSO interaction.
The T11 operator corresponds to the spin-other-orbit interaction,
  and the t11 operator (associated with electrostatically-correlated
   spin-orbit) originates from configuration interaction analysis.
  To their sum a factor proportional to the operator z13 is
  subtracted since its effect is redundant to the spin-orbit
  interaction. The factor of 1/6 is not on Judd's 1968 paper, but it
   is on \"Chen, Xueyuan, Guokui Liu, Jean Margerie, and Michael F
  Reid. \"A Few Mistakes in Widely Used Data Files for Fn
  Configurations Calculations.\" Journal of Luminescence 128, no. 3
  (2008): 421-27\".
The values for the reduced matrix elements of z13 are obtained from
   Table IX of the same paper. The value for a13 is from table VIII.
Rigorously speaking the Pk parameters here are subscripted. The
  conversion to superscripted parameters is performed elsewhere with
   the Prescaling replacement rules.
";
ReducedSOOandECSOinf2[SL_, SpLp_]  := Module[
  {a13, z13, z13inf2, matElement, redSOOandECSOinf2},
  (
    a13 = (-33 M0 + 3 M2 + 15/11 M4 -
          6 P0 + 3/2 (35 P2 + 77 P4 + 143 P6));
    z13inf2 = <|
        {"1S","3P"} -> 2,
        {"3P","3P"} -> 1,
        {"3P","1D"} -> -Sqrt[(15/2)],
        {"1D","3F"} -> Sqrt[10],
        {"3F","3F"} -> Sqrt[14],
        {"3F","1G"} -> -Sqrt[11],
        {"1G","3H"} -> Sqrt[10],
        {"3H","3H"} -> Sqrt[55],
        {"3H","1I"} -> -Sqrt[(13/2)]
        |>;
    matElement = Which[
      MemberQ[Keys[z13inf2],{SL,SpLp}],
        z13inf2[{SL,SpLp}],
      MemberQ[Keys[z13inf2],{SpLp,SL}],
        z13inf2[{SpLp,SL}],
      True,
        0
    ];
    redSOOandECSOinf2 = (
        ReducedT11inf2[SL, SpLp] +
        Reducedt11inf2[SL, SpLp] -
        a13 / 6 * matElement
    );
    redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
    Return[redSOOandECSOinf2];
  )
];

ReducedSOOandECSOinfn::usage = "ReducedSOOandECSOinfn[numE, SL,
  SpLp] calculates the reduced matrix elements of the (spin-other-
  orbit + ECSO) operator for the f^numE configuration corresponding
  to the terms SL and SpLp. This is done recursively, starting from
  tabulated values for f^2 from \"Judd, BR, HM Crosswhite, and
  Hannah Crosswhite. \"Intra-Atomic Magnetic Interactions for f
  Electrons.\" Physical Review 169, no. 1 (1968): 130.\", and by
  using equation (4) of that same paper.
";
ReducedSOOandECSOinfn[numE_, SL_, SpLp_] := Module[
```

```
    {spin, orbital, t, S, L, Sp, Lp,
    idx1, idx2, cfpSL, cfpSpLp, parentSL,
    Sb, Lb, Sbp, Lbp, parentSpLp, funval},
    (
      {spin, orbital} = {1/2, 3};
      {S, L}   = FindSL[SL];
      {Sp, Lp} = FindSL[SpLp];
      t = 1;
      cfpSL    = CFP[{numE, SL}];
      cfpSpLp  = CFP[{numE, SpLp}];
      funval   = Sum[
        (
          parentSL = cfpSL[[idx2, 1]];
          parentSpLp = cfpSpLp[[idx1, 1]];
          {Sb, Lb}   = FindSL[parentSL];
          {Sbp, Lbp} = FindSL[parentSpLp];
          phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
          (
            phase *
            cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
            SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
            SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
            SOOandECSOLSTable[{numE - 1, parentSL, parentSpLp}]
          )
        ),
      {idx1, 2, Length[cfpSpLp]},
      {idx2, 2, Length[cfpSL]}
      ];
      funval *= numE / (numE - 2) * Sqrt[TPO[S, Sp, L, Lp]];
      Return[funval];
    )
  ];

  GenerateSOOandECSOLSTable::usage = "GenerateSOOandECSOLSTable[nmax]
     generates the LS reduced matrix elements of the spin-other-orbit
    + ECSO for the f^n configurations up to n=nmax. The values for n=1
     and n=2 are taken from \"Judd, BR, HM Crosswhite, and Hannah
    Crosswhite. \"Intra-Atomic Magnetic Interactions for f Electrons.\
    " Physical Review 169, no. 1 (1968): 130.\", and the values for n
    >2 are calculated recursively using equation (4) of that same
    paper. The values are then exported to a file \"
    ReducedSOOandECSOLSTable.m\" in the data folder of this module.
    The values are also returned as an association.";
  Options[GenerateSOOandECSOLSTable] = {"Progress" -> True, "Export"
    -> True};
  GenerateSOOandECSOLSTable[nmax_Integer, OptionsPattern[]] := (
    If[And[OptionValue["Progress"], frontEndAvailable],
      (
        numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
    numE]]^2, {numE, 1, nmax}]];
        counters   = Association[Table[numE->0, {numE, 1, nmax}]];
        totalIters = Total[Values[numItersai[[1;;nmax]]]];
        template1  = StringTemplate["Iteration `numiter` of `
    totaliter`"];
        template2  = StringTemplate["`remtime` min remaining"];
    template3 = StringTemplate["Iteration speed = `speed` ms/it"];
        template4  = StringTemplate["Time elapsed = `runtime` min"];
        progBar = PrintTemporary[
          Dynamic[
            Pane[
              Grid[{
                    {Superscript["f", numE]},
                    {template1[<|"numiter"->numiter, "totaliter"->
    totalIters|>]},
                    {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                    {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
    ]], 0.1]|>]},
                    {template3[<|"speed"->Round[QuantityMagnitude[Now
    -startTime, "ms"]/(numiter), 0.01]|>]}, {ProgressIndicator[Dynamic
    [numiter], {1, totalIters}]}
                  },
                  Frame->All
            ],
            Full,
```

```
1669                Alignment->Center
1670              ]
1671            ]
1672          ];
1673        )
1674      ];
1675      SOOandECSOLSTable = <||>;
1676      numiter   = 1;
1677      startTime = Now;
1678      Do[
1679        (
1680          numiter+= 1;
1681          SOOandECSOLSTable[{numE, SL, SpLp}] = Which[
1682            numE==1,
1683            0,
1684            numE==2,
1685            SimplifyFun[ReducedSOOandECSOinf2[SL, SpLp]],
1686            True,
1687            SimplifyFun[ReducedSOOandECSOinfn[numE,  SL, SpLp]]
1688          ];
1689        ),
1690      {numE, 1, nmax},
1691      {SL, AllowedNKSLTerms[numE]},
1692      {SpLp, AllowedNKSLTerms[numE]}
1693      ];
1694      If[And[OptionValue["Progress"], frontEndAvailable],
1695        NotebookDelete[progBar]];
1696      If[OptionValue["Export"],
1697        (fname = FileNameJoin[{moduleDir, "data", "
      ReducedSOOandECSOLSTable.m"}];
1698        Export[fname, SOOandECSOLSTable];
1699        )
1700      ];
1701      Return[SOOandECSOLSTable];
1702    );
1703
1704    (* ################## Reduced SOO and ECSO ################### *)
1705    (* ############################################################ *)
1706
1707    (* ############################################################ *)
1708    (* ###################### Spin-Spin ########################### *)
1709
1710    ReducedT22inf2::usage = "ReducedT22inf2[SL, SpLp] returns the
        reduced matrix element of the scalar component of the double
        tensor T22 for the terms SL, SpLp in f^2.
1711    Data used here for m0, m2, m4 is from Table I of Judd, BR, HM
        Crosswhite, and Hannah Crosswhite. Intra-Atomic Magnetic
        Interactions for f Electrons. Physical Review 169, no. 1 (1968):
        130.
1712    ";
1713    ReducedT22inf2[SL_, SpLp_] := Module[
1714      {statePosition, PsiPsipStates, m0, m2, m4, Tkk2m},
1715      (
1716        T22inf2 = <|
1717        {"3P", "3P"} -> -12 M0 - 24 M2 - 300/11 M4,
1718        {"3P", "3F"} -> 8/Sqrt[3] (3 M0 + M2 - 100/11 M4),
1719        {"3F", "3F"} -> 4/3 Sqrt[14] (-M0 + 8 M2 - 200/11 M4),
1720        {"3F", "3H"} -> 8/3 Sqrt[11/2] (2 M0 - 23/11 M2 - 325/121 M4),
1721        {"3H", "3H"} -> 4/3 Sqrt[143] (M0 - 34/11 M2 - (1325/1573) M4)
1722        |>;
1723        Which[
1724          MemberQ[Keys[T22inf2],{SL,SpLp}],
1725            Return[T22inf2[{SL,SpLp}]],
1726          MemberQ[Keys[T22inf2],{SpLp,SL}],
1727            Return[T22inf2[{SpLp,SL}]],
1728          True,
1729            Return[0]
1730        ]
1731      )
1732    ];
1733
1734    ReducedT22infn::usage = "ReducedT22infn[n, SL, SpLp] calculates the
         reduced matrix element of the T22 operator for the f^n
        configuration corresponding to the terms SL and SpLp.
1735    This is done by using equation (4) of \"Judd, BR, HM Crosswhite,
        and Hannah Crosswhite. \"Intra-Atomic Magnetic Interactions for f
```

```
    Electrons.\" Physical Review 169, no. 1 (1968): 130.\"
1736  ";
1737  ReducedT22infn[numE_, SL_, SpLp_] := Module[
1738    {spin, orbital, t, idx1, idx2, S, L,
1739    Sp, Lp, cfpSL, cfpSpLp, parentSL,
1740    parentSpLp, Sb, Lb, Tnkk, phase, Sbp, Lbp},
1741    (
1742      {spin, orbital} = {1/2, 3};
1743      {S, L}   = FindSL[SL];
1744      {Sp, Lp} = FindSL[SpLp];
1745      t = 2;
1746      cfpSL    = CFP[{numE, SL}];
1747      cfpSpLp  = CFP[{numE, SpLp}];
1748      Tnkk = Sum[(
1749        parentSL   = cfpSL[[idx2, 1]];
1750        parentSpLp = cfpSpLp[[idx1, 1]];
1751        {Sb, Lb}   = FindSL[parentSL];
1752        {Sbp, Lbp} = FindSL[parentSpLp];
1753        phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
1754        (
1755          phase *
1756          cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
1757          SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
1758          SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
1759          T22Table[{numE - 1, parentSL, parentSpLp}]
1760        )
1761      ),
1762      {idx1, 2, Length[cfpSpLp]},
1763      {idx2, 2, Length[cfpSL]}
1764      ];
1765      Tnkk *= numE / (numE - 2) * Sqrt[TPO[S,Sp,L,Lp]];
1766      Return[Tnkk];
1767    )
1768  ];
1769
1770  GenerateT22Table::usage = "GenerateT22Table[nmax] generates the LS
       reduced matrix elements for the double tensor operator T22 in f^n
       up to n=nmax. If the option \"Export\" is set to true then the
       resulting association is saved to the data folder. The values for
       n=1 and n=2 are taken from \"Judd, BR, HM Crosswhite, and Hannah
       Crosswhite. \"Intra-Atomic Magnetic Interactions for f Electrons.\
       " Physical Review 169, no. 1 (1968): 130.\", and the values for n
       >2 are calculated recursively using equation (4) of that same
       paper.
1771  This is an intermediate step to the calculation of the reduced
       matrix elements of the spin-spin operator.";
1772  Options[GenerateT22Table] = {"Export" -> True, "Progress" -> True};
1773  GenerateT22Table[nmax_Integer, OptionsPattern[]] := (
1774    If[And[OptionValue["Progress"], frontEndAvailable],
1775      (
1776        numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
       numE]]^2, {numE, 1, nmax}]];
1777        counters = Association[Table[numE->0, {numE, 1, nmax}]];
1778        totalIters = Total[Values[numItersai[[1;;nmax]]]];
1779        template1 = StringTemplate["Iteration `numiter` of `totaliter
       `"];
1780        template2 = StringTemplate["`remtime` min remaining"];
       template3 = StringTemplate["Iteration speed = `speed` ms/it"];
1781        template4 = StringTemplate["Time elapsed = `runtime` min"];
1782        progBar = PrintTemporary[
1783          Dynamic[
1784            Pane[
1785              Grid[{{Superscript["f", numE]},
1786                    {template1[<|"numiter"->numiter, "totaliter"->
       totalIters|>]},
1787                    {template4[<|"runtime"->Round[QuantityMagnitude[
       UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
1788                    {template2[<|"remtime"->Round[QuantityMagnitude[
       UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
       ]], 0.1]|>]},
1789                    {template3[<|"speed"->Round[QuantityMagnitude[Now
       -startTime, "ms"]/(numiter), 0.01]|>]},
1790                    {ProgressIndicator[Dynamic[numiter], {1,
       totalIters}]}},
1791                    Frame->All],
1792                    Full,
```

```
                    Alignment->Center]
                ]
              ];
    )
  ];
  T22Table = <||>;
  startTime = Now;
  numiter = 1;
  Do[
    (
      numiter+= 1;
      T22Table[{numE, SL, SpLp}] = Which[
        numE==1,
        0,
        numE==2,
        SimplifyFun[ReducedT22inf2[SL, SpLp]],
        True,
        SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
      ];
    ),
  {numE, 1, nmax},
  {SL,   AllowedNKSLTerms[numE]},
  {SpLp, AllowedNKSLTerms[numE]}
  ];
  If[And[OptionValue["Progress"],frontEndAvailable],
    NotebookDelete[progBar]
  ];
  If[OptionValue["Export"],
    (
      fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"}];
      Export[fname, T22Table];
    )
  ];
  Return[T22Table];
);

SpinSpin::usage = "SpinSpin[n, SL, SpLp, J] returns the matrix element <|SL,J|spin-spin|SpLp,J|> for the spin-spin operator within the configuration f^n. This matrix element is independent of MJ. This is obtained by querying the relevant reduced matrix element from the association T22Table, putting in the adequate phase, and 6-j symbol.
This is calculated according to equation (3) in \"Judd, BR, HM Crosswhite, and Hannah Crosswhite. \"Intra-Atomic Magnetic Interactions for f Electrons.\" Physical Review 169, no. 1 (1968): 130.\"
\".
";
SpinSpin[numE_, SL_, SpLp_, J_] := Module[
  {S, L, Sp, Lp, α, val},
  (
    α = 2;
    {S, L} = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            T22Table[{numE, SL, SpLp}]
          );
    Return[val]
  )
];

GenerateSpinSpinTable::usage = "GenerateSpinSpinTable[nmax] generates the reduced matrix elements in the |LSJ> basis for the spin-spin operator. It returns an association where the keys are of the form {numE, SL, SpLp, J}. If the option \"Export\" is set to True then the resulting object is saved to the data folder. Since this is a scalar operator, there is no MJ dependence. This dependence only comes into play when the crystal field contribution is taken into account.";
Options[GenerateSpinSpinTable] = {"Export"->False};
GenerateSpinSpinTable[nmax_, OptionsPattern[]] :=
  (
    SpinSpinTable = <||>;
```

```
      PrintTemporary[Dynamic[numE]];
      Do[
        SpinSpinTable[{numE, SL, SpLp, J}] = (SpinSpin[numE, SL, SpLp
    , J]);,
      {numE, 1, nmax},
      {J, MinJ[numE], MaxJ[numE]},
      {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
      {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
      ];
      If[OptionValue["Export"],
      (fname = FileNameJoin[{moduleDir, "data", "SpinSpinTable.m"}];
        Export[fname, SpinSpinTable];
        )
      ];
      Return[SpinSpinTable];
    );

(* ###################### Spin-Spin ######################### *)
(* ############################################################ *)

(*
  ######################################################################
   *)
(* ## Spin-Other-Orbit and Electrostatically-Correlated-Spin-Orbit
  ## *)

SOOandECSO::usage = "SOOandECSO[n, SL, SpLp, J] returns the matrix
  element <|SL,J|spin-spin|SpLp,J|> for the combined effects of the
  spin-other-orbit interaction and the electrostatically-correlated-
  spin-orbit (which originates from configuration interaction
  effects) within the configuration f^n. This matrix element is
  independent of MJ. This is obtained by querying the relevant
  reduced matrix element by querying the association
  SOOandECSOLSTable and putting in the adequate phase and 6-j symbol
  . The SOOandECSOLSTable puts together the reduced matrix elements
  from three operators.
This is calculated according to equation (3) in \"Judd, BR, HM
  Crosswhite, and Hannah Crosswhite. \"Intra-Atomic Magnetic
  Interactions for f Electrons.\" Physical Review 169, no. 1 (1968):
   130.\".
";
SOOandECSO[numE_, SL_, SpLp_, J_] := Module[
  {S, Sp, L, Lp, α, val},
  (
    α = 1;
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    val = (
            Phaser[Sp + L + J] *
            SixJay[{Sp, Lp, J}, {L, S, α}] *
            SOOandECSOLSTable[{numE, SL, SpLp}]
          );
    Return[val];
  )
];

Prescaling = {P2 -> P2/225, P4 -> P4/1089, P6 -> 25 * P6 / 184041};

GenerateSOOandECSOTable::usage = "GenerateSOOandECSOTable[nmax]
  generates the reduced matrix elements in the |LSJ> basis for the (
  spin-other-orbit + electrostatically-correlated-spin-orbit)
  operator. It returns an association where the keys are of the form
   {n, SL, SpLp, J}. If the option \"Export\" is set to True then
  the resulting object is saved to the data folder. Since this is a
  scalar operator, there is no MJ dependence. This dependence only
  comes into play when the crystal field contribution is taken into
  account.";
Options[GenerateSOOandECSOTable] = {"Export"->False};
GenerateSOOandECSOTable[nmax_, OptionsPattern[]] := (
  SOOandECSOTable = <||>;
  Do[
    SOOandECSOTable[{numE, SL, SpLp, J}] = (SOOandECSO[numE, SL,
  SpLp, J] /. Prescaling);,
  {numE, 1, nmax},
  {J, MinJ[numE], MaxJ[numE]},
  {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
```

```
    {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
    ];
    If[OptionValue["Export"],
    (
      fname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
      Export[fname, SOOandECSOTable];
    )
    ];
    Return[SOOandECSOTable];
  );

  (* ## Spin-Other-Orbit and Electrostatically-Correlated-Spin-Orbit
    ## *)
  (*
    ######################################################################
     *)

  (* ############################################################ *)
  (* ################### Magnetic Interactions ################## *)

  MagneticInteractions::usage = "MagneticInteractions[{numE, SL, SLp,
     J}] returns the matrix element of the magnetic interaction
    between the terms SL and SLp in the f^numE configuration for the
    given value of J. The interaction is given by the sum of the spin-
    spin, the spin-other-orbit,  and the electrostatically-correlated-
    spin-orbit interactions.
  The part corresponding to the spin-spin interaction is provided by
    SpinSpin[{numE, SL, SLp, J}].
  The part corresponding to SOO and ECSO is provided by the function
    SOOandECSO[{numE, SL, SLp, J}].
  The option \"ChenDeltas\" can be used to include or exclude the
    Chen deltas from the calculation. The default is to exclude them.
    If this option is used, then the chenDeltas association needs to
    be loaded into the session with LoadChenDeltas[].";
  Options[MagneticInteractions] = {"ChenDeltas" -> False};
  MagneticInteractions[{numE_, SL_, SLp_, J_}, OptionsPattern[]] :=
    Module[
    {key, ss, sooandecso, total,
    S, L, Sp, Lp, phase, sixjay,
    M0v, M2v, M4v,
    P2v, P4v, P6v},
    (
      key        = {numE, SL, SLp, J};
      ss         = \[Sigma]SS * SpinSpinTable[key];
      sooandecso = SOOandECSOTable[key];
      total = ss + sooandecso;
      total = SimplifyFun[total];
      If[
        Not[OptionValue["ChenDeltas"]],
        Return[total]
      ];
      (* In the type A errors the wrong values are different *)
      If[MemberQ[Keys[chenDeltas["A"]], {numE, SL, SLp}],
        (
          {S, L}   = FindSL[SL];
          {Sp, Lp} = FindSL[SLp];
          phase    = Phaser[Sp + L + J];
          sixjay   = SixJay[{Sp, Lp, J}, {L, S, 1}];
          {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SL,
     SLp}]["wrong"];
          total    = (
                          phase * sixjay *
                          (
                            M0v*M0 + M2v*M2 + M4v*M4 +
                            P2v*P2 + P4v*P4 + P6v*P6
                          )
                      );
          total    = wChErrA * total + (1 - wChErrA) * (ss +
    sooandecso)
        )
      ];
      (* In the type B errors the wrong values are zeros all around
    *)
      If[MemberQ[chenDeltas["B"], {numE, SL, SLp}],
        (
          total  = (1 - wChErrB) * (ss + sooandecso)
```

```
        )
      ];
      Return[total];
    )
  ];

  (* ################## Magnetic Interactions ################## *)
  (* ########################################################### *)

  (* ########################################################### *)
  (* ##################### Free-Ion Energies ##################### *)

  GenerateFreeIonTable::usage="GenerateFreeIonTable[] generates an
    association for free-ion energies in terms of Slater integrals Fk
    and spin-orbit parameter ζ. It returns an association where the
    keys are of the form {nE, SL, SpLp}. If the option \"Export\" is
    set to True then the resulting object is saved to the data folder.
     The free-ion Hamiltonian is the sum of the electrostatic and spin
    -orbit interactions. The electrostatic interaction is given by the
     function Electrostatic[{numE, SL, SpLp}] and the spin-orbit
    interaction is given by the function SpinOrbitTable[{numE, SL,
    SpLp}]. The values for the electrostatic interaction are taken
    from the data file ElectrostaticTable.m and the values for the
    spin-orbit interaction are taken from the data file SpinOrbitTable
    .m. The values for the free-ion Hamiltonian are then exported to a
     file \"FreeIonTable.m\" in the data folder of this module. The
    values are also returned as an association.";
  Options[GenerateFreeIonTable] = {"Export" -> False};
  GenerateFreeIonTable[OptionsPattern[]] := Module[
    {terms, numEH, zetaSign, fname, FreeIonTable},
    (
      If[Not[ValueQ[ElectrostaticTable]],
        LoadElectrostatic[]
        ];
      If[Not[ValueQ[SpinOrbitTable]],
        LoadSpinOrbit[]
      ];
      If[Not[ValueQ[ReducedUkTable]],
        LoadUk[]
      ];
      FreeIonTable = <||>;
      Do[
        (
          terms = AllowedNKSLJTerms[nE];
          numEH = Min[nE, 14 - nE];
          zetaSign = If[nE > 7, -1, 1];
          Do[
            FreeIonTable[{nE,term[[1]],term[[2]]}] = (
              Electrostatic[{numEH, term[[1]], term[[1]]}] +
                zetaSign * SpinOrbitTable[{numEH, term[[1]], term
    [[1]], term[[2]]}]
              ),
            {term, terms}];
        ),
        {nE, 1, 14}
      ];
      If[OptionValue["Export"],
        (
          fname = FileNameJoin[{moduleDir, "data", "FreeIonTable.m"
    }];
          Export[fname, FreeIonTable];
        )
      ];
      Return[FreeIonTable];
    )
  ];

  LoadFreeIon::usage = "LoadFreeIon[] loads the free-ion energies
    from the data folder. The values are stored in the association
    FreeIonTable.";
  LoadFreeIon[] := (
    If[ValueQ[FreeIonTable],
      Return[]
    ];
    PrintTemporary["Loading the association of free-ion energies ..."
    ];
```

```
    FreeIonTableFname = FileNameJoin[{moduleDir, "data", "FreeIonTable.m"}];
    FreeIonTable = If[!FileExistsQ[FreeIonTableFname],
      (
        PrintTemporary[">> FreeIonTable.m not found, generating ..."];
        GenerateFreeIonTable["Export" -> True]
      ),
        Import[FreeIonTableFname]
    ];
  );

  (* ##################### Free-Ion Energies ##################### *)
  (* ############################################################ *)

  (* ############################################################ *)
  (* ###################### Crystal Field ####################### *)

  Cqk::usage = "Cqk[numE, q, k, NKSL, J, M, NKSLp, Jp, Mp]. In Wybourne (1965) see equations 6-3, 6-4, and 6-5. Also in TASS see equation 11.53.";
  Cqk[numE_, q_, k_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_] := Module[
    {S, Sp, L, Lp, orbital, val},
    (
      orbital = 3;
      {S, L}   = FindSL[NKSL];
      {Sp, Lp} = FindSL[NKSLp];
      f1  = ThreeJay[{J, -M}, {k, q}, {Jp, Mp}];
      val =
        If[f1==0,
          0,
          (
            f2 = SixJay[{L, J, S}, {Jp, Lp, k}] ;
            If[f2==0,
              0,
              (
                f3 = ReducedUkTable[{numE, orbital, NKSL, NKSLp, k}];
                If[f3==0,
                  0,
                  (
                    (
                      Phaser[J - M + S + Lp + J + k] *
                      Sqrt[TPO[J, Jp]] *
                      f1 *
                      f2 *
                      f3 *
                      Ck[orbital, k]
                    )
                  )
                ]
              )
            ]
          )
        ];
      Return[val];
    )
  ];

  Bqk::usage = "Real part of the Bqk coefficients.";
  Bqk[q_, 2] := {B02/2, B12, B22}[[q + 1]];
  Bqk[q_, 4] := {B04/2, B14, B24, B34, B44}[[q + 1]];
  Bqk[q_, 6] := {B06/2, B16, B26, B36, B46, B56, B66}[[q + 1]];

  Sqk::usage = "Imaginary part of the Bqk coefficients.";
  Sqk[q_, 2] := {0, S12, S22}[[q + 1]];
  Sqk[q_, 4] := {0, S14, S24, S34, S44}[[q + 1]];
  Sqk[q_, 6] := {0, S16, S26, S36, S46, S56, S66}[[q + 1]];

  CrystalField::usage = "CrystalField[n, NKSL, J, M, NKSLp, Jp, Mp] calculates the matrix element of the crystal field in terms of Bqk and Sqk parameters for configuration f^numE. It is calculated as an association with keys of the form {n, NKSL, J, M, NKSLp, Jp, Mp}.";
  CrystalField[numE_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_] := (
    Sum[
      (
```

```
        cqk  = Cqk[numE,  q, k, NKSL, J, M, NKSLp, Jp, Mp];
        cmqk = Cqk[numE, -q, k, NKSL, J, M, NKSLp, Jp, Mp];
        Bqk[q, k]   * (cqk + (-1)^q * cmqk) +
        I*Sqk[q, k] * (cqk - (-1)^q * cmqk)
        ),
    {k, {2, 4, 6}},
    {q, 0, k}
    ]
)

TotalCFIters::usage = "TotalIters[i, j] returns total number of
  function evaluations for calculating all the matrix elements for
  the \!\(\*SuperscriptBox[\(f\), \(i\)]\) to the \!\(\*
  SuperscriptBox[\(f\), \(j\)]\) configurations.";
TotalCFIters[i_, j_] := (
  numIters = {196, 8281, 132496, 1002001, 4008004, 9018009,
  11778624};
  Return[Total[numIters[[i ;; j]]]];
  )

GenerateCrystalFieldTable::usage = "GenerateCrystalFieldTable[{
  numEs}] computes the matrix values for the crystal field
  interaction for f^n configurations the given list of numE in
  numEs. The function calculates the association CrystalFieldTable
  with keys of the form {numE, NKSL, J, M, NKSLp, Jp, Mp}. If the
  option \"Export\" is set to True, then the result is exported to
  the data subfolder for the folder in which this package is in. If
  the option \"Progress\" is set to True then an interactive
  progress indicator is shown. If \"Compress\" is set to true the
  exported values are compressed when exporting.";
Options[GenerateCrystalFieldTable] = {"Export" -> False, "Progress"
   -> True, "Compress" -> True, "Overwrite" -> False};
GenerateCrystalFieldTable[numEs_List:{1,2,3,4,5,6,7},
  OptionsPattern[]] := (
  ExportFun =
  If[OptionValue["Compress"],
    ExportMZip,
    Export
  ];
  numiter = 1;
  template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
  template2 = StringTemplate["`remtime` min remaining"];
  template3 = StringTemplate["Iteration speed = `speed` ms/it"];
  template4 = StringTemplate["Time elapsed = `runtime` min"];
  totalIter = Total[TotalCFIters[#, #] & /@ numEs];
  freebies = 0;
  startTime = Now;
  If[And[OptionValue["Progress"], frontEndAvailable],
    progBar = PrintTemporary[
      Dynamic[
        Pane[
          Grid[
            {
              {Superscript["f", numE]},
              {template1[<|"numiter" -> numiter, "totaliter" ->
  totalIter|>]},
              {template4[<|"runtime" -> Round[QuantityMagnitude[
  UnitConvert[(Now - startTime), "min"]], 0.1]|>]},
              {template2[<|"remtime" -> Round[QuantityMagnitude[
  UnitConvert[(Now - startTime)/(numiter - freebies) * (totalIter -
  numiter), "min"]], 0.1]|>]},
              {template3[<|"speed" -> Round[QuantityMagnitude[Now -
   startTime, "ms"]/(numiter-freebies), 0.01]|>]},
              {ProgressIndicator[Dynamic[numiter], {1, totalIter}]}
            },
          Frame -> All
          ],
        Full,
        Alignment -> Center
        ]
      ]
    ];
  ];
  Do[
    (
      exportFname = FileNameJoin[{moduleDir, "data", "
```

```
    CrystalFieldTable_f"<>ToString[numE]<>".m"}];
2144        If[And[FileExistsQ[exportFname],Not[OptionValue["Overwrite"
    ]]],
2145          Echo["File exists, skipping ..."];
2146          numiter+=  TotalCFIters[numE, numE];
2147          freebies+= TotalCFIters[numE, numE];
2148          Continue[];
2149        ];
2150        CrystalFieldTable = <||>;
2151        Do[
2152          (
2153            numiter+=1;
2154            {S,L}   = FindSL[NKSL];
2155            {Sp,Lp} = FindSL[NKSLp];
2156            CrystalFieldTable[{numE,NKSL,J,M,NKSLp,Jp,Mp}] =
2157              Which[
2158                Abs[M-Mp]>6,
2159                0,
2160                Abs[S-Sp]!=0,
2161                0,
2162                True,
2163                CrystalField[numE,NKSL,J,M,NKSLp,Jp,Mp]
2164              ];
2165          ),
2166        {J,  MinJ[numE], MaxJ[numE]},
2167        {Jp, MinJ[numE], MaxJ[numE]},
2168        {M,  AllowedMforJ[J]},
2169        {Mp, AllowedMforJ[Jp]},
2170        {NKSL, First/@AllowedNKSLforJTerms[numE,J]},
2171        {NKSLp, First/@AllowedNKSLforJTerms[numE,Jp]}
2172        ];
2173        If[And[OptionValue["Progress"],frontEndAvailable],
2174          NotebookDelete[progBar]
2175        ];
2176        If[OptionValue["Export"],
2177          (
2178            Echo["Exporting to file " <> ToString[exportFname]];
2179            ExportFun[exportFname, CrystalFieldTable];
2180          )
2181        ];
2182      ),
2183    {numE, numEs}
2184    ]
2185  )
2186
2187  ParseBenelli2015::usage="ParseBenelli2015[] parses the data from
      file /data/benelli_and_gatteschi_table3p3.csv which corresponds to
       Table 3.3 of Benelli and Gatteshi, Introduction to Molecular.
      This data provides the form that the crystal field has under
      different point group symmetries. This function parses that data
      into an association with keys equal to strings representing any of
       the 32 crystallographic point groups.";
2188  Options[ParseBenelli2015] = {"Export"->False};
2189  ParseBenelli2015[OptionsPattern[]] := Module[
2190    {fname, crystalSym,
2191    crystalSymmetries, parseFun,
2192    chars, qk, groupName, family,
2193    groupNum, params},
2194  (
2195    fname      = FileNameJoin[{moduleDir,"data","
      benelli_and_gatteschi_table3p3.csv"}];
2196    crystalSym = Import[fname][[2;;33]];
2197    crystalSymmetries = <||>;
2198    parseFun[txt_] := (
2199      chars = Characters[txt];
2200      qk    = chars[[-2;;]];
2201      If[chars[[1]]=="R",
2202      (
2203        Return[{ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}]}]
2204      ),
2205      (
2206        If[qk[[1]]=="0",
2207          Return[{ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}]}]
2208        ];
2209        Return[{
2210          ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}],
```

```
          ToExpression@StringJoin[{"S",qk[[1]],qk[[2]]}]
        }]
      )]
    );
    Do[
    (
      groupNum  = Round@ToExpression@row[[1]];
      groupName = row[[2]];
      family    = row[[3]];
      params    = Select[row[[4;;]],And[FreeQ[#,Missing],#=!=""]&];
      params    = parseFun/@params;
      params    = <|"BqkSqk"->Sort@Flatten[params],
        "aliases"->{groupNum},
        "constraints"->{}|>;
      If[MemberQ[{"T","Th","O","Td","Oh"},groupName],
        params["constraints"] = {
            {B44 ->  Sqrt[5/14] B04, B46 -> -Sqrt[7/2] B06},
            {B44 -> -Sqrt[5/14] B04, B46 ->  Sqrt[7/2] B06}
            }
      ];
      If[StringContainsQ[groupName,","],
      (
        {alias1, alias2}          = StringSplit[groupName,","];
        crystalSymmetries[alias1] = params;
        crystalSymmetries[alias1]["aliases"] = {groupNum, alias2};
        crystalSymmetries[alias2] = params;
        crystalSymmetries[alias2]["aliases"] = {groupNum, alias1};
      ),
      (
      crystalSymmetries[groupName] = params;
      )
      ]
    ),
    {row,crystalSym}];
    crystalSymmetries["source"] = "Benelli and Gatteschi, 2015,
    Introduction to Molecular Magnetism, table 3.3.";
    If[OptionValue["Export"],
      Export[FileNameJoin[{moduleDir,"data","
    crystalFieldFunctionalForms.m"}],crystalSymmetries];
    ];
    Return[crystalSymmetries];
  )
  ];

  CrystalFieldForm::usage = "CrystalFieldForm[symmetryGroup] returns
    an association that describes the crystal field parameters that
    are necessary to describe a crystal field for the given symmetry
    group.

  The symmetry group must be given as a string in Schoenflies
    notation and must be one of C1, Ci, S2, Cs, C1h, C2, C2h, C2v, D2,
     D2h, S4, C4, C4h, D2d, C4v, D4, D4h, C3, S6, C3h, C3v, D3, D3d,
    D3h, C6, C6h, C6v, D6, D6h, T, Th, Td, O, Oh.

  The returned association has three keys:
    \"BqkSqk\" whose values is a list with the nonzero Bqk and Sqk
    parameters;
    \"constraints\" whose value is either an empty list, or a lists
    of replacements rules that are constraints on the Bqk and Sqk
    parameters;
    \"simplifier\" whose value is an association that can be used to
    set to zero the crystal field parameters that are zero for the
    given symmetry group;
    \"aliases\" whose value is a list with the integer by which the
    point group is also known for and an alternate Schoenflies symbol
    if it exists.

  This uses data from table 3.3 in Benelli and Gatteschi, 2015.";
  CrystalFieldForm[symmetryGroupString_] := (
    If[Not@ValueQ[crystalFieldFunctionalForms],
      crystalFieldFunctionalForms = Import[FileNameJoin[{moduleDir, "
    data", "crystalFieldFunctionalForms.m"}]];
    ];
    cfForm = crystalFieldFunctionalForms[symmetryGroupString];
    simplifier = Association[(# -> 0) &/@ Complement[cfSymbols,
    cfForm["BqkSqk"]]];
```

```
    Return[Join[cfForm, <|"simplifier"->simplifier|>]];
  )

  (* ###################### Crystal Field ######################### *)
  (* ############################################################## *)

  (* ############################################################## *)
  (* #### Configuration-Interaction via Casimir Operators ###### *)

  CasimirSO3::usage = "CasimirSO3[{SL, SpLp}] returns LS reduced
    matrix element of the configuration interaction term corresponding
     to the Casimir operator of R3.";
  CasimirSO3[{SL_, SpLp_}] := (
    {S, L} = FindSL[SL];
    If[SL == SpLp,
      α * L * (L + 1),
      0
    ]
  )

  GG2U::usage = "GG2U is an association whose keys are labels for the
     irreducible representations of group G2 and whose values are the
    eigenvalues of the corresponding Casimir operator.
  Reference: Wybourne, \"Spectroscopic Properties of Rare Earths\",
    table 2-6.";
  GG2U = Association[{
      "00" -> 0,
      "10" -> 6/12 ,
      "11" -> 12/12,
      "20" -> 14/12,
      "21" -> 21/12,
      "22" -> 30/12,
      "30" -> 24/12,
      "31" -> 32/12,
      "40" -> 36/12}
    ];

  CasimirG2::usage = "CasimirG2[{SL, SpLp}] returns LS reduced matrix
     element of the configuration interaction term corresponding to
    the Casimir operator of G2.";
  CasimirG2[{SL_, SpLp_}] := (
    Ulabel = FindNKLSTerm[SL][[1]][[4]];
    If[SL==SpLp,
      β * GG2U[Ulabel],
      0
    ]
  )

  GSO7W::usage = "GSO7W is an association whose keys are labels for
    the irreducible representations of group R7 and whose values are
    the eigenvalues of the corresponding Casimir operator.
  Reference: Wybourne, \"Spectroscopic Properties of Rare Earths\",
    table 2-7.";
  GSO7W := Association[
    {
      "000" -> 0,
      "100" -> 3/5,
      "110" -> 5/5,
      "111" -> 6/5,
      "200" -> 7/5,
      "210" -> 9/5,
      "211" -> 10/5,
      "220" -> 12/5,
      "221" -> 13/5,
      "222" -> 15/5
    }
  ];

  CasimirSO7::usage = "CasimirSO7[{SL, SpLp}] returns the LS reduced
    matrix element of the configuration interaction term corresponding
     to the Casimir operator of R7.";
  CasimirSO7[{SL_, SpLp_}] := (
    Wlabel = FindNKLSTerm[SL][[1]][[3]];
    If[SL==SpLp,
      γ * GSO7W[Wlabel],
      0
```

```
    ]
  )

  ElectrostaticConfigInteraction::usage = "
    ElectrostaticConfigInteraction[numE, {SL, SpLp}] returns the
    matrix element for configuration interaction as approximated by
    the Casimir operators of the groups R3, G2, and R7. SL and SpLp
    are strings that represent terms under LS coupling.";
  ElectrostaticConfigInteraction[numE_, {SL_, SpLp_}] := Module[
    {S, L, val},
    (
      If[
        Or[numE == 1, numE==13],
        Return[0];
      ];
      {S, L} = FindSL[SL];
      val = (
        If[SL == SpLp,
          CasimirSO3[{SL, SL}] +
          CasimirSO7[{SL, SL}] +
          CasimirG2[{SL, SL}],
          0
        ]
        );
      ElectrostaticConfigInteraction[numE, {S, L}] = val;
      Return[val];
    )
  ];

  (* #### Configuration-Interaction via Casimir Operators ###### *)
  (* ############################################################ *)

  (* ############################################################ *)
  (* ###################### Block assembly ###################### *)

  JJBlockMatrix::usage = "For given J, J' in the f^n configuration
    JJBlockMatrix[numE, J, J'] determines all the SL S'L' terms that
    may contribute to them and using those it provides the matrix
    elements <J, LS | H | J', LS'>. H having contributions from the
    following interactions: Coulomb, spin-orbit, spin-other-orbit,
    electrostatically-correlated-spin-orbit, spin-spin, three-body
    interactions, and crystal-field.";
  Options[JJBlockMatrix] = {"Sparse"->True, "ChenDeltas"->False};
  JJBlockMatrix[numE_, J_, Jp_, CFTable_, OptionsPattern[]] := Module
    [
    {NKSLJMs, NKSLJMps, NKSLJM, NKSLJMp,
    SLterm, SpLpterm,
    MJ, MJp,
    subKron, matValue, eMatrix},
    (
      NKSLJMs  = AllowedNKSLJMforJTerms[numE, J];
      NKSLJMps = AllowedNKSLJMforJTerms[numE, Jp];
      eMatrix =
        Table[
          (*Condition for a scalar matrix op*)
          SLterm   = NKSLJM[[1]];
          SpLpterm = NKSLJMp[[1]];
          MJ       = NKSLJM[[3]];
          MJp      = NKSLJMp[[3]];
          subKron  = (
              KroneckerDelta[J, Jp] *
              KroneckerDelta[MJ, MJp]
                    );
          matValue =
            If[subKron==0,
              0,
                (
                  ElectrostaticTable[{numE, SLterm, SpLpterm}] +
                  ElectrostaticConfigInteraction[numE, {SLterm,
    SpLpterm}] +
                  SpinOrbitTable[{numE, SLterm, SpLpterm, J}] +
                  MagneticInteractions[{numE, SLterm, SpLpterm, J},
                    "ChenDeltas" -> OptionValue["ChenDeltas"]] +
                  ThreeBodyTable[{numE, SLterm, SpLpterm}]
                )
            ];
```

```
2398            matValue += CFTable[{numE, SLterm, J, MJ, SpLpterm, Jp, MJp
    }];
2399            matValue,
2400          {NKSLJMp, NKSLJMps},
2401          {NKSLJM , NKSLJMs}
2402          ];
2403       If[OptionValue["Sparse"],
2404         eMatrix = SparseArray[eMatrix]
2405       ];
2406       Return[eMatrix]
2407     )
2408     ];
2409
2410     EnergyStates::usage = "Alias for AllowedNKSLJMforJTerms. At some
        point may be used to redefine states used in basis.";
2411     EnergyStates[numE_, J_]:= AllowedNKSLJMforJTerms[numE, J];
2412
2413     JJBlockMatrixFileName::usage = "JJBlockMatrixFileName[numE] gives
        the filename for the energy matrix table for an atom with numE f-
        electrons. The function admits an optional parameter \"
        FilenameAppendix\" which can be used to modify the filename.";
2414     Options[JJBlockMatrixFileName] = {"FilenameAppendix" -> ""};
2415     JJBlockMatrixFileName[numE_Integer, OptionsPattern[]] := (
2416       fileApp = OptionValue["FilenameAppendix"];
2417       fname = FileNameJoin[{moduleDir,
2418           "hams",
2419           StringJoin[{"f", ToString[numE], "_JJBlockMatrixTable",
      fileApp ,".m"}]}];
2420       Return[fname];
2421     );
2422
2423     TabulateJJBlockMatrixTable::usage = "TabulateJJBlockMatrixTable[
        numE, CFTable] returns an association JJBlockMatrixTable with keys
         equal to lists of the form {numE, J, Jp} and values equal to JJ
        blocks of the semi-empirical Hamiltonian. The function admits an
        optional parameter \"Sparse\" which can be used to control whether
         the matrix is returned as a SparseArray or not. The default is
        True. Another admitted option is \"ChenDeltas\" which can be used
        to include or exclude the Chen deltas from the calculation. The
        default is to exclude them. If this option is used, then the
        chenDeltas association needs to be loaded into the session with
        LoadChenDeltas[].";
2424     Options[TabulateJJBlockMatrixTable] = {"Sparse"->True, "ChenDeltas"
      ->False};
2425     TabulateJJBlockMatrixTable[numE_, CFTable_, OptionsPattern[]] := (
2426       JJBlockMatrixTable = <||>;
2427       totalIterations = Length[AllowedJ[numE]]^2;
2428       template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
2429       template2 = StringTemplate["`remtime` min remaining"];
2430       template4 = StringTemplate["Time elapsed = `runtime` min"];
2431       numiter   = 0;
2432       startTime = Now;
2433       If[$FrontEnd =!= Null,
2434         (
2435           temp = PrintTemporary[
2436             Dynamic[
2437               Grid[
2438                 {
2439                   {template1[<|"numiter"->numiter, "totaliter"->
      totalIterations|>]},
2440                   {template2[<|"remtime"->Round[QuantityMagnitude[
      UnitConvert[(Now-startTime)/(Max[1,numiter])*(totalIterations-
      numiter), "min"]], 0.1]|>]},
2441                   {template4[<|"runtime"->Round[QuantityMagnitude[
      UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
2442                   {ProgressIndicator[numiter, {1, totalIterations}]}
2443                 }
2444               ]
2445           ]
2446           ];
2447         )
2448       ];
2449       Do[
2450         (
2451           JJBlockMatrixTable[{numE, J, Jp}] = JJBlockMatrix[numE, J, Jp
      , CFTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas" ->
```

```
    OptionValue["ChenDeltas"]];
2452        numiter += 1;
2453      ),
2454    {Jp, AllowedJ[numE]},
2455    {J, AllowedJ[numE]}
2456    ];
2457    If[$FrontEnd =!= Null,
2458      NotebookDelete[temp]
2459    ];
2460    Return[JJBlockMatrixTable];
2461  );
2462
2463  TabulateManyJJBlockMatrixTables::usage = "
      TabulateManyJJBlockMatrixTables[{n1, n2, ...}] calculates the
      tables of matrix elements for the requested f^n_i configurations.
      The function does not return the matrices themselves. It instead
      returns an association whose keys are numE and whose values are
      the filenames where the output of TabulateJJBlockMatrixTables was
      saved to.The output consists of an association whose keys are of
      the form {n, J, Jp} and whose values are rectangular arrays given
      the values of <|LSJMJa|H|L'S'J'MJ'a'|>.";
2464  Options[TabulateManyJJBlockMatrixTables] = {"Overwrite"->False, "
      Sparse"->True, "ChenDeltas"->False, "FilenameAppendix"-> "", "
      Compressed" -> False};
2465  TabulateManyJJBlockMatrixTables[ns_, OptionsPattern[]] := (
2466    overwrite = OptionValue["Overwrite"];
2467    fNames = <||>;
2468    fileApp = OptionValue["FilenameAppendix"];
2469    ExportFun = If[OptionValue["Compressed"], ExportMZip, Export];
2470    Do[
2471      (
2472        CFdataFilename = FileNameJoin[{moduleDir, "data", "
      CrystalFieldTable_f"<>ToString[numE]<>".zip"}];
2473        PrintTemporary["Importing CrystalFieldTable from ",
      CFdataFilename, " ..."];
2474        CrystalFieldTable = ImportMZip[CFdataFilename];
2475
2476        PrintTemporary["#------- numE = ", numE, " -------#"];
2477        exportFname = JJBlockMatrixFileName[numE, "FilenameAppendix"
      -> fileApp];
2478        fNames[numE] = exportFname;
2479        If[FileExistsQ[exportFname] && Not[overwrite],
2480          Continue[]
2481        ];
2482        JJBlockMatrixTable = TabulateJJBlockMatrixTable[numE,
      CrystalFieldTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas"
      -> OptionValue["ChenDeltas"]];
2483        If[FileExistsQ[exportFname] && overwrite,
2484          DeleteFile[exportFname]
2485        ];
2486        ExportFun[exportFname, JJBlockMatrixTable];
2487
2488        ClearAll[CrystalFieldTable];
2489      ),
2490    {numE, ns}
2491    ];
2492    Return[fNames];
2493  );
2494
2495  EffectiveHamiltonian::usage = "EffectiveHamiltonian[numE] returns
      the Hamiltonian matrix for the f^numE configuration. The matrix is
       returned as a SparseArray.
2496  The function admits an optional parameter \"FilenameAppendix\"
      which can be used to control which variant of the JJBlocks is used
       to assemble the matrix.
2497  It also admits an optional parameter \"IncludeZeeman\" which can be
       used to include the Zeeman interaction. The default is False.
2498  The option \"Set t2Switch\" can be used to toggle on or off setting
       the t2 selector automatically or not, the default is True, which
      replaces the parameter according to numE.
2499  The option \"ReturnInBlocks\" can be used to return the matrix in
      block or flattened form. The default is to return it in flattened
      form.";
2500  Options[EffectiveHamiltonian] = {
2501        "FilenameAppendix" -> "",
2502        "IncludeZeeman"    -> False,
```

```
        "Set t2Switch"     -> True,
        "ReturnInBlocks"   -> False,
        "OperatorBasis"    -> "Legacy"};
  EffectiveHamiltonian[nf_, OptionsPattern[]] := Module[
    {numE, ii, jj, howManyJs, Js, blockHam, opBasis},
    (
      (*######################################*)
      ImportFun = ImportMZip;
      opBasis = OptionValue["OperatorBasis"];
      If[Not[MemberQ[{"Legacy", "MostlyOrthogonal", "Orthogonal"},
    opBasis]],
        Echo["Operator basis " <> opBasis <> " not recognized, using
    \"Legacy\" basis."];
        opBasis = "Legacy";
      ];
      If[opBasis == "Orthogonal",
        Echo["Operator basis \"Orthogonal\" not implemented yet,
    aborting ..."];
        Return[Null];
      ];
      (*######################################*)
      If[opBasis == "MostlyOrthogonal",
        (
          blockHam = EffectiveHamiltonian[nf,
          "OperatorBasis" -> "Legacy",
          "FilenameAppendix" -> OptionValue["FilenameAppendix"],
          "IncludeZeeman" -> OptionValue["IncludeZeeman"],
          "Set t2Switch" -> OptionValue["Set t2Switch"],
          "ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
          paramChanger = Which[
            nf < 7,
            <|
              F0 -> 1/91 (54 E1p+91 E0p+78 γp),
              F2 -> (15/392 *
                (
                  140   E1p  +
                  20020 E2p +
                  1540  E3p  +
                  770 αp -
                  70 γp  +
                  22 Sqrt[2] T2p -
                  11 Sqrt[2] nf T2p t2Switch -
                  11 Sqrt[2] (14-nf) T2p (1 - t2Switch)
                )
              ),
              F4 -> (99/490 *
                (
                  70   E1p -
                  9100 E2p +
                  280  E3p +
                  140 αp -
                  35 γp  +
                  4 Sqrt[2] T2p -
                  2 Sqrt[2] nf T2p t2Switch -
                  2 Sqrt[2] (14-nf) T2p (1-t2Switch)
                )
                ),
              F6 -> (5577/7000 *
                (
                  20  E1p +
                  700 E2p -
                  140 E3p -
                  70 αp -
                  10 γp -
                  2 Sqrt[2] T2p +
                  Sqrt[2] nf T2p t2Switch +
                  Sqrt[2] (14-nf) T2p (1-t2Switch)
                )
                ),
              ζ  -> ζ,
              α -> (5 αp)/4,
              β  -> -6  (5 αp + βp),
              γ -> 5/2 (2 βp + 5 γp),
              T2 -> 0
            |>,
          nf >= 7,
```

```
            <|
              F0 -> 1/91 (54 E1p+91 E0p+78 γp),
              F2 -> (15/392 *
                (
                  140   E1p  +
                  20020 E2p +
                  1540  E3p  +
                  770 αp -
                  70 γp  +
                  22 Sqrt[2] T2p -
                  11 Sqrt[2] nf T2p
                )
              ),
              F4 -> (99/490 *
                (
                  70   E1p -
                  9100 E2p +
                  280  E3p +
                  140 αp -
                  35 γp  +
                  4 Sqrt[2] T2p -
                  2 Sqrt[2] nf T2p
                )
                ),
              F6 -> (5577/7000 *
                (
                  20  E1p +
                  700 E2p -
                  140 E3p -
                  70 αp -
                  10 γp -
                  2 Sqrt[2] T2p +
                  Sqrt[2] nf T2p
                )
                ),
              ζ  -> ζ,
              α -> (5 αp)/4,
              β  -> -6  (5 αp + βp),
              γ -> 5/2 (2 βp + 5 γp),
              T2 -> 0
            |>
          ];
          blockHamM0 = Which[
            OptionValue["ReturnInBlocks"] == False,
            ReplaceInSparseArray[blockHam, paramChanger],
            OptionValue["ReturnInBlocks"] == True,
            Map[ReplaceInSparseArray[#, paramChanger]&, blockHam,
{2}]
          ];
          Return[blockHamM0];
        )
      ];
      (*######################################*)

      (*hole-particle equivalence enforcement*)
      numE = nf;
      allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2,
T2p,
        T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
        α, β, γ, B02, B04, B06, B12, B14, B16,
        B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16,
S22,
        S24, S26, S34, S36, S44, S46, S56, S66, T11p, T12, T14, T15,
T16,
        T17, T18, T19, Bx, By, Bz};
      params0 = AssociationThread[allVars, allVars];
      If[nf > 7,
        (
          numE = 14 - nf;
          params = HoleElectronConjugation[params0];
          If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
        ),
        params = params0;
        If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
      ];
      (* Load symbolic expressions for LS,J,J' energy sub-matrices.
```

```
    *)
2648        emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
    OptionValue["FilenameAppendix"]];
2649        JJBlockMatrixTable = ImportFun[emFname];
2650        (*Patch together the entire matrix representation using J,J'
    blocks.*)
2651        PrintTemporary["Patching JJ blocks ..."];
2652        Js         = AllowedJ[numE];
2653        howManyJs = Length[Js];
2654        blockHam  = ConstantArray[0, {howManyJs, howManyJs}];
2655        Do[
2656          blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[
    jj]]}];,
2657        {ii, 1, howManyJs},
2658        {jj, 1, howManyJs}
2659        ];
2660        (* Once the block form is created flatten it *)
2661        If[Not[OptionValue["ReturnInBlocks"]],
2662          (blockHam = ArrayFlatten[blockHam];
2663          blockHam  = ReplaceInSparseArray[blockHam, params];
2664          ),
2665          (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
    ,{2}];)
2666        ];
2667
2668        If[OptionValue["IncludeZeeman"],
2669          (
2670            PrintTemporary["Including Zeeman terms ..."];
2671            {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
    ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
2672            blockHam += - teslaToKayser * (Bx * magx + By * magy + Bz *
     magz);
2673          )
2674        ];
2675        Return[blockHam];
2676      )
2677  ];
2678
2679  SimplerEffectiveHamiltonian::usage = "SimplerEffectiveHamiltonian[
      numE, simplifier] is a simple addition to EffectiveHamiltonian
      that applies a given simplification to the full Hamiltonian.
      simplifier is a list of replacement rules.
2680  If the option \"Export\" is set to True, then the function also
      exports the resulting sparse array to the ./hams/ folder.
2681  The option \"PrependToFilename\" can be used to prepend a string to
       the filename to which the function may export to.
2682  The option \"Return\" can be used to choose whether the function
      returns the matrix or not.
2683  The option \"Overwrite\" can be used to overwrite the file if it
      already exists, if this options is set to False then this function
       simply reloads a file that it assumed to be present already in
      the ./hams folder.
2684  The option \"IncludeZeeman\" can be used to toggle the inclusion of
       the Zeeman interaction with an external magnetic field.
2685  The option \"OperatorBasis\" can be used to choose the basis in
      which the operator is expressed. The default is the \"Legacy\"
      basis. Order alternatives being: \"MostlyOrthogonal\" and \"
      Orthogonal\". In the \"Legacy\" alternative the operators used are
       the same as in Carnall's work. In the \"MostlyOrthogonal\"  all
      operators are orthogonal except those corresponding to the Mk and
      Pk parameters. In the \"Orthogonal\" basis all operators are
      orthogonal, with the operators corresponding to the Mk and Pk
      parameters replaced by zi operators and accompanying ai
      coefficients. The \"Orthogonal\" option has not been implemented
      yet.";
2686  Options[SimplerEffectiveHamiltonian] = {
2687    "Export"->True,
2688    "PrependToFilename"->"",
2689    "EorF"->"F",
2690    "Overwrite" -> False,
2691    "Return" -> True,
2692    "Set t2Switch" -> False,
2693    "IncludeZeeman" -> False,
2694    "OperatorBasis" -> "Legacy"
2695    };
2696  SimplerEffectiveHamiltonian[numE_, simplifier_, OptionsPattern[]]
```

```
    := Module[
2697  {thisHam,fname, fnamemx},
2698  (
2699    If[Not[ValueQ[ElectrostaticTable]],
2700      LoadElectrostatic[]
2701      ];
2702    If[Not[ValueQ[SOOandECSOTable]],
2703      LoadSOOandECSO[]
2704    ];
2705    If[Not[ValueQ[SpinOrbitTable]],
2706      LoadSpinOrbit[]
2707    ];
2708    If[Not[ValueQ[SpinSpinTable]],
2709      LoadSpinSpin[]
2710    ];
2711    If[Not[ValueQ[ThreeBodyTable]],
2712      LoadThreeBody[]
2713    ];
2714
2715    opBasis = OptionValue["OperatorBasis"];
2716    If[Not[MemberQ[{"Legacy", "MostlyOrthogonal", "Orthogonal"},
    opBasis]],
2717      Echo["Operator basis " <> opBasis <> " not recognized, using
    \"Legacy\" basis."];
2718      opBasis = "Legacy";
2719    ];
2720    If[opBasis == "Orthogonal",
2721      Echo["Operator basis \"Orthogonal\" not implemented yet,
    aborting ..."];
2722      Return[Null];
2723    ];
2724
2725    fnamePrefix = Which[
2726      opBasis == "Legacy",
2727      "SymbolicMatrix-f",
2728      opBasis == "MostlyOrthogonal",
2729      "SymbolicMatrix-mostly-orthogonal-f",
2730      opBasis == "Orthogonal",
2731      "SymbolicMatrix-orthogonal-f"
2732    ];
2733    fname   = FileNameJoin[{moduleDir, "hams",
2734      OptionValue["PrependToFilename"] <> fnamePrefix <> ToString[
    numE] <> ".m"}];
2735    fnamemx = FileNameJoin[{moduleDir, "hams",
2736      OptionValue["PrependToFilename"] <> fnamePrefix <> ToString[
    numE] <> ".mx"}];
2737    fnamezip = FileNameJoin[{moduleDir, "hams",
2738      OptionValue["PrependToFilename"] <> fnamePrefix <> ToString[
    numE] <> ".zip"}];
2739    If[Or[FileExistsQ[fname],
2740          FileExistsQ[fnamemx],
2741          FileExistsQ[fnamezip]]
2742      && Not[OptionValue["Overwrite"]],
2743      (
2744        If[OptionValue["Return"],
2745          (
2746            Which[
2747              FileExistsQ[fnamezip],
2748              (
2749                Echo["File " <> fnamezip <> " already exists, and
    option \"Overwrite\" is set to False, loading file ..."];
2750                thisHam = ImportMZip[fnamezip];
2751                Return[thisHam];
2752              ),
2753              FileExistsQ[fnamemx],
2754              (
2755                Echo["File " <> fnamemx <> " already exists, and
    option \"Overwrite\" is set to False, loading file ..."];
2756                thisHam = Import[fnamemx];
2757                Return[thisHam];
2758              ),
2759              FileExistsQ[fname],
2760              (
2761                Echo["File " <> fname <> " already exists, and
    option \"Overwrite\" is set to False, loading file ..."];
2762                thisHam = Import[fname];
```

```
2763                    Echo["Exporting to file " <> fnamemx <> " for
    quicker loading."];
2764                    Export[fnamemx, thisHam];
2765                    Return[thisHam];
2766                  )
2767                ]
2768              ),
2769              (
2770                Echo["File " <> fname <> " already exists, skipping ...
    "];
2771                Return[Null];
2772              )
2773            ]
2774          )
2775        ];
2776
2777        thisHam = EffectiveHamiltonian[numE, "Set t2Switch" ->
    OptionValue["Set t2Switch"], "IncludeZeeman" -> OptionValue["
    IncludeZeeman"], "OperatorBasis" -> opBasis];
2778        If[Length[simplifier] > 0,
2779          thisHam = ReplaceInSparseArray[thisHam, simplifier];
2780        ];
2781        (* This removes zero entries from being included in the sparse
    array *)
2782        thisHam = SparseArray[thisHam];
2783        If[OptionValue["Export"],
2784        (
2785          If[Not@FileExistsQ[fname],
2786          (
2787            Echo["Exporting to file " <> fname];
2788            Export[fname,   thisHam];
2789          )
2790          ];
2791          If[Not@FileExistsQ[fnamemx],
2792          (
2793            Echo["Exporting to file " <> fnamemx];
2794            Export[fnamemx, thisHam];
2795          )
2796          ];
2797        )
2798        ];
2799        If[OptionValue["Return"],
2800          Return[thisHam],
2801          Return[Null]
2802        ];
2803      )
2804    ];
2805
2806    ScalarLSJMFromLS::usage = "ScalarLSJMFromLS[numE,
      LSReducedMatrixElements]. Given the LS-reduced matrix elements
      LSReducedMatrixElements of a scalar operator, this function
      returns the corresponding LSJM representation. This is returned as
       a SparseArray.";
2807    ScalarLSJMFromLS[numE_, LSReducedMatrixElements_] := Module[
2808      {jjBlocktable, NKSLJMs, NKSLJMps, J, Jp, eMatrix, SLterm,
      SpLpterm,
2809      MJ, MJp, subKron, matValue, Js, howManyJs, blockHam, ii, jj},
2810      (
2811      SparseDiagonalArray[diagonalElements_] := SparseArray[
2812        Table[{i, i} -> diagonalElements[[i]],
2813          {i, 1, Length[diagonalElements]}]
2814        ];
2815      SparseZeroArray[width_, height_] := (
2816        SparseArray[
2817          Join[
2818          Table[{1, i} -> 0, {i, 1, width}],
2819          Table[{i, 1} -> 0, {i, 1, height}]
2820          ]
2821          ]
2822        );
2823      jjBlockTable = <||>;
2824      Do[
2825        NKSLJMs  = AllowedNKSLJMforJTerms[numE, J];
2826        NKSLJMps = AllowedNKSLJMforJTerms[numE, Jp];
2827        If[J != Jp,
2828          jjBlockTable[{numE, J, Jp}] = SparseZeroArray[Length[NKSLJMs
```

```
  ], Length[NKSLJMps]];
    Continue[];
    ];
    eMatrix = Table[
      (* Condition for a scalar matrix op *)
      SLterm = NKSLJM[[1]];
      SpLpterm = NKSLJMp[[1]];
      MJ = NKSLJM[[3]];
      MJp = NKSLJMp[[3]];
      subKron = (KroneckerDelta[MJ, MJp]);
      matValue = If[subKron == 0,
      0,
        (
          Which[MemberQ[Keys[LSReducedMatrixElements], {numE,
 SLterm, SpLpterm}],
            LSReducedMatrixElements[{numE, SLterm, SpLpterm}],
            MemberQ[Keys[LSReducedMatrixElements], {numE, SpLpterm,
  SLterm}],
            LSReducedMatrixElements[{numE, SpLpterm, SLterm}],
            True,
            0
          ]
        )
      ];
      matValue,
      {NKSLJMp, NKSLJMps},
      {NKSLJM, NKSLJMs}
      ];
    jjBlockTable[{numE, J, Jp}] = SparseArray[eMatrix],
    {J, AllowedJ[numE]},
    {Jp, AllowedJ[numE]}
    ];

   Js = AllowedJ[numE];
   howManyJs = Length[Js];
   blockHam = ConstantArray[0, {howManyJs, howManyJs}];
   Do[blockHam[[jj, ii]] = jjBlockTable[{numE, Js[[ii]], Js[[jj
 ]]}];,
    {ii, 1, howManyJs},
    {jj, 1, howManyJs}];
   blockHam = ArrayFlatten[blockHam];
   blockHam = SparseArray[blockHam];
   Return[blockHam];
   )
   ];

(* ###################### Block assembly ###################### *)
(* ############################################################ *)

(* ############################################################ *)
(* ##################### Level Description ##################### *)

FreeHam::usage = "FreeHam[JJBlocks, numE] given the JJ blocks of
  the Hamiltonian for f^n, this function returns a list with all the
   scalar-simplified versions of the blocks.";
FreeHam[JJBlocks_List, numE_Integer] := Module[
  {Js, basisJ, pivot, freeHam, idx, J,
  thisJbasis, shrunkBasisPositions, theBlock},
  (
    Js      = AllowedJ[numE];
    basisJ  = BasisLSJMJ[numE, "AsAssociation" -> True];
    pivot   = If[OddQ[numE], 1/2, 0];
    freeHam = Table[(
      J          = Js[[idx]];
      theBlock   = JJBlocks[[idx]];
      thisJbasis = basisJ[J];
      (* find the basis vectors that end with pivot *)
      shrunkBasisPositions = Flatten[Position[thisJbasis, {_ ...,
 pivot}]];
      (* take only those rows and columns *)
      theBlock[[shrunkBasisPositions, shrunkBasisPositions]]
    ),
    {idx, 1, Length[Js]}
    ];
    Return[freeHam];
  )
```

```
];

ListRepeater::usage = "ListRepeater[list, reps] repeats each
  element of list reps times.";
ListRepeater[list_List, repeats_Integer] := (
  Flatten[ConstantArray[#, repeats] & /@ list]
);

ListLever::usage = "ListLever[vecs, multiplicity] takes a list of
  vectors and returns all interleaved shifted versions of them.";
ListLever[vecs_, multiplicity_] := Module[
{uppityVecs, uppityVec},
(
  uppityVecs = Table[(
    uppityVec = PadRight[{#}, multiplicity] & /@ vec;
    uppityVec = Permutations /@ uppityVec;
    uppityVec = Transpose[uppityVec];
    uppityVec = Flatten /@ uppityVec
    ),
  {vec, vecs}
  ];
  Return[Flatten[uppityVecs, 1]];
)
];

EigenLever::usage = "EigenLever[eigenSys, multiplicity] takes a
  list eigenSys of the form {eigenvalues, eigenvectors} and returns
  the eigenvalues repeated multiplicity times and the eigenvectors
  interleaved and shifted accordingly.";
EigenLever[eigenSys_, multiplicity_] := Module[
  {eigenVals, eigenVecs,
  leveledEigenVecs, leveledEigenVals},
  (
    {eigenVals, eigenVecs} = eigenSys;
    leveledEigenVals       = ListRepeater[eigenVals, multiplicity];
    leveledEigenVecs       = ListLever[eigenVecs, multiplicity];
    Return[{Flatten[leveledEigenVals], leveledEigenVecs}]
  )
];


LevelSimplerEffectiveHamiltonian::usage = "
  LevelSimplerEffectiveHamiltonian[numE] is a variation of
  EffectiveHamiltonian that returns the diagonal JJ Hamiltonian
  blocks applying a simplifier and with simplifications adequate for
   the level description. The keys of the given association
  correspond to the different values of J that are possible for f^
  numE, the values are sparse array that are meant to be interpreted
   in the basis provided by BasisLSJ.
The option \"Simplifier\" is a list of symbols that are set to zero
  . At a minimum this has to include the crystal field parameters.
  By default this includes everything except the Slater parameters
  Fk and the spin orbit coupling ζ.
The option \"Export\" controls whether the resulting association is
   saved to disk, the default is True and the resulting file is
  saved to the ./hams/ folder. A hash is appended to the filename
  that corresponds to the simplifier used in the resulting
  expression. If the option \"Overwrite\" is set to False then these
   files may be used to quickly retrieve a previously computed case.
   The file is saved both in .m and .mx format.
The option \"PrependToFilename\" can be used to append a string to
  the filename to which the function may export to.
The option \"Return\" can be used to choose whether the function
  returns the matrix or not.
The option \"Overwrite\" can be used to overwrite the file if it
  already exists.";
Options[LevelSimplerEffectiveHamiltonian] = {
  "Export" -> True,
  "PrependToFilename" -> "",
  "Overwrite" -> False,
  "Return" -> True,
  "Simplifier" -> Join[
    {F0,\[Sigma]SS},
    cfSymbols,
    TSymbols,
    casimirSymbols,
```

```
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
  ]
};
LevelSimplerEffectiveHamiltonian[numE_Integer, OptionsPattern[]] :=
  Module[
  {thisHamAssoc, Js, fname,
  fnamemx, hash, simplifier},
  (
    simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
    hash       = Hash[simplifier];
    If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
    If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
    If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
    If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
    If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
    fname   = FileNameJoin[{moduleDir,"hams",OptionValue["
  PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
  <>ToString[hash]<>".m"}];
    fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
  PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
  <>ToString[hash]<>".mx"}];
    If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue
  ["Overwrite"]],
    (
      If[OptionValue["Return"],
      (
        Which[FileExistsQ[fnamemx],
        (
          Echo["File " <> fnamemx <> " already exists, and option \
  "Overwrite\" is set to False, loading file ..."];
          thisHamAssoc=Import[fnamemx];
          Return[thisHamAssoc];
        ),
        FileExistsQ[fname],
        (
          Echo["File " <> fname <> " already exists, and option \"
  Overwrite\" is set to False, loading file ..."];
          thisHamAssoc=Import[fname];
          Echo["Exporting to file " <> fnamemx <> " for quicker
  loading."];
          Export[fnamemx,thisHamAssoc];
          Return[thisHamAssoc];
        )
        ]
      ),
      (
        Echo["File " <> fname <> " already exists, skipping ..."];
        Return[Null];
      )
      ]
    )
    ];
    Js           = AllowedJ[numE];
    thisHamAssoc = EffectiveHamiltonian[numE,
      "Set t2Switch"->True,
      "IncludeZeeman"->False,
      "ReturnInBlocks"->True
    ];
    thisHamAssoc = Diagonal[thisHamAssoc];
    thisHamAssoc = Map[SparseArray[ReplaceInSparseArray[#,
  simplifier]]&,thisHamAssoc,{1}];
    thisHamAssoc = FreeHam[thisHamAssoc,numE];
    thisHamAssoc = AssociationThread[Js->thisHamAssoc];
    If[OptionValue["Export"],
      (
        Echo["Exporting to file " <> fname <> " and to " <> fnamemx
  ];
        Export[fname,thisHamAssoc];
        Export[fnamemx,thisHamAssoc];
      )
    ];
    If[OptionValue["Return"],
      Return[thisHamAssoc],
      Return[Null]
```

```
3015        ];
3016      )
3017    ];
3018
3019    LevelSolver::usage = "LevelSolver[numE, params] puts together (or
          retrieves from disk) the symbolic level Hamiltonian for the f^numE
           configuration and solves it for the given params returning the
          resultant energies and eigenstates.
3020    If the option \"Return as states\" is set to False, then the
          function returns an association whose keys are values for J in f^
          numE, and whose values are lists with two elements. The first
          element being equal to the ordered basis for the corresponding
          subspace, given as a list of lists of the form {LS string, J}. The
           second element being another list of two elements, the first
          element being equal to the energies and the second being equal to
          the corresponding normalized eigenvectors. The energies given have
           been subtracted the energy of the ground state.
3021    If the option \"Return as states\" is set to True, then the
          function returns a list with three elements. The first element is
          the global level basis for the f^numE configuration, given as a
          list of lists of the form {LS string, J}. The second element are
          the mayor LSJ components in the returned eigenstates. The third
          element is a list of lists with three elements, in each list the
          first element being equal to the energy, the second being equal to
           the value of J, and the third being equal to the corresponding
          normalized eigenvector (given as a row). The energies given have
          been subtracted the energy of the ground state, and the states
          have been sorted in order of increasing energy.
3022    The following options are admitted:
3023      - \"Overwrite Hamiltonian\", if set to True the function will
          overwrite the symbolic Hamiltonian. Default is False.
3024      - \"Return as states\", see description above. Default is True.
3025      - \"Simplifier\", this is a list with symbols that are set to
          zero for defining the parameters kept in the level  description.
3026    ";
3027    Options[LevelSolver] = {
3028      "Overwrite Hamiltonian" -> False,
3029      "Return as states" -> True,
3030      "Simplifier" -> Join[
3031        cfSymbols,
3032        TSymbols,
3033        casimirSymbols,
3034        pseudoMagneticSymbols,
3035        marvinSymbols,
3036        DeleteCases[magneticSymbols,ζ]
3037      ],
3038      "PrintFun" -> PrintTemporary
3039    };
3040    LevelSolver[numE_Integer, params0_Association, OptionsPattern[]] :=
           Module[
3041      {ln, simplifier, simpleHam, basis,
3042      numHam, eigensys, startTime, endTime,
3043      diagonalTime, params=params0, globalBasis,
3044      eigenVectors, eigenEnergies, eigenJs,
3045      states, groundEnergy, allEnergies, PrintFun},
3046      (
3047        ln         = theLanthanides[[numE]];
3048        basis      = BasisLSJ[numE, "AsAssociation"->True];
3049        simplifier = OptionValue["Simplifier"];
3050        PrintFun   = OptionValue["PrintFun"];
3051        PrintFun["> LevelSolver for ",ln," with ",numE," f-electrons."
          ];
3052        PrintFun["> Loading the symbolic level Hamiltonian ..."];
3053        simpleHam  = LevelSimplerEffectiveHamiltonian[numE,
3054          "Simplifier" -> simplifier,
3055          "Overwrite"  -> OptionValue["Overwrite Hamiltonian"]
3056        ];
3057        (* Everything that is not given is set to zero *)
3058        PrintFun["> Setting to zero every parameter not given ..."];
3059        params     = ParamPad[params, "PrintFun" -> PrintFun];
3060        PrintFun[params];
3061        (* Create the numeric hamiltonian *)
3062        PrintFun["> Replacing parameters in the J-blocks of the
          Hamiltonian to produce numeric arrays ..."];
3063        numHam     = N /@ Map[ReplaceInSparseArray[#, params]&,
          simpleHam];
```

```
3064      Clear[simpleHam];
3065      (* Eigensolver *)
3066      PrintFun["> Diagonalizing the numerical Hamiltonian within each
          separate J-subspace ..."];
3067      startTime    = Now;
3068      eigensys     = Eigensystem /@ numHam;
3069      endTime      = Now;
3070      diagonalTime = QuantityMagnitude[endTime-startTime,"Seconds"];
3071      allEnergies  = Flatten[First/@Values[eigensys]];
3072      groundEnergy = Min[allEnergies];
3073      eigensys     = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys
         ];
3074      eigensys     = Association@KeyValueMap[#1->{basis[#1],#2} &,
         eigensys];
3075      PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
3076      If[OptionValue["Return as states"],
3077        (
3078          PrintFun["> Padding the eigenvectors to correspond to the
         level basis ..."];
3079          eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values
         [#[[2, 2]] & /@ eigensys]];
3080          globalBasis   = Flatten[Values[basis], 1];
3081          eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
3082          eigenJs       = Flatten[KeyValueMap[ConstantArray[#1,
         Length[#2[[2, 2]]]] &, eigensys]];
3083          states        = Transpose[{eigenEnergies, eigenJs,
         eigenVectors}];
3084          states        = SortBy[states, First];
3085          eigenVectors  = Last /@ states;
3086          LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
         InputForm[#[[2]]]]) & /@ globalBasis;
3087          majorComponentIndices = Ordering[Abs[#]][[-1]] & /@
         eigenVectors;
3088          levelLabels           = LSJmultiplets[[
         majorComponentIndices]];
3089          Return[{globalBasis, levelLabels, states}];
3090        ),
3091        Return[{basis, eigensys}]
3092      ];
3093    )
3094  ];
3095
3096
3097  (* ##################### Level Description ##################### *)
3098  (* ############################################################ *)
3099
3100  (* ############################################################ *)
3101  (* ##################### Optical Operators ##################### *)
3102
3103  magOp = <||>;
3104
3105  JJBlockMagDip::usage = "JJBlockMagDip[numE, J, Jp] returns an array
       for the LSJM matrix elements of the magnetic dipole operator
      between states with given J and Jp. The option \"Sparse\" can be
      used to return a sparse matrix. The default is to return a sparse
      matrix.
3106  See eqn 15.7 in TASS.
3107  Here it is provided in atomic units in which the Bohr magneton is
      1/2.
3108  \[Mu] = -(1/2) (L + gs S)
3109  We are using the Racah convention for the reduced matrix elements
      in the Wigner-Eckart theorem. See TASS eqn 11.15.
3110  ";
3111  Options[JJBlockMagDip]={"Sparse"->True};
3112  JJBlockMagDip[numE_, braJ_, ketJ_, OptionsPattern[]] := Module[
3113    {braSLJs, ketSLJs,
3114    braSLJ,   ketSLJ,
3115    braSL,    ketSL,
3116    braS,     braL,
3117    ketS,     ketL,
3118    braMJ,    ketMJ,
3119    matValue, magMatrix,
3120    summand1, summand2,
3121    threejays},
3122    (
3123      braSLJs  = AllowedNKSLJMforJTerms[numE,braJ];
```

```
3124        ketSLJs  = AllowedNKSLJMforJTerms[numE,ketJ];
3125        magMatrix  = Table[
3126          braSL     = braSLJ[[1]];
3127          ketSL     = ketSLJ[[1]];
3128          {braS, braL}   = FindSL[braSL];
3129          {ketS, ketL}   = FindSL[ketSL];
3130          braMJ       = braSLJ[[3]];
3131          ketMJ       = ketSLJ[[3]];
3132          summand1    = If[Or[braJ  != ketJ,
3133                              braSL != ketSL],
3134            0,
3135            Sqrt[braJ*(braJ+1)*TPO[braJ]]
3136          ];
3137          (* looking at the string includes checking L=L', S=S', and \
    alpha=\alpha'*)
3138          summand2 = If[braSL!= ketSL,
3139            0,
3140            (gs-1) *
3141              Phaser[braS+braL+ketJ+1] *
3142                Sqrt[TPO[braJ]*TPO[ketJ]] *
3143                  SixJay[{braJ,1,ketJ},{braS,braL,braS}] *
3144                    Sqrt[braS(braS+1)TPO[braS]]
3145          ];
3146          matValue = summand1 + summand2;
3147          (* We are using the Racah convention for red matrix elements
    in Wigner-Eckart *)
3148          threejays = (ThreeJay[{braJ, -braMJ}, {1, #}, {ketJ, ketMJ}]
    &) /@ {-1,0,1};
3149          threejays *= Phaser[braJ-braMJ];
3150          matValue  = - 1/2 * threejays * matValue;
3151          matValue,
3152        {braSLJ, braSLJs},
3153        {ketSLJ, ketSLJs}
3154        ];
3155        If[OptionValue["Sparse"],
3156          magMatrix = SparseArray[magMatrix]
3157        ];
3158        Return[magMatrix];
3159      )
3160    ];
3161
3162    Options[TabulateJJBlockMagDipTable]={"Sparse"->True};
3163    TabulateJJBlockMagDipTable[numE_,OptionsPattern[]] := (
3164      JJBlockMagDipTable=<||>;
3165      Js=AllowedJ[numE];
3166      Do[
3167        (
3168          JJBlockMagDipTable[{numE,braJ,ketJ}]=
3169            JJBlockMagDip[numE,braJ,ketJ,"Sparse"->OptionValue["Sparse"
    ]]
3170        ),
3171      {braJ, Js},
3172      {ketJ, Js}
3173      ];
3174      Return[JJBlockMagDipTable];
3175    );
3176
3177    TabulateManyJJBlockMagDipTables::usage = "
      TabulateManyJJBlockMagDipTables[{n1, n2, ...}] calculates the
      tables of matrix elements for the requested f^n_i configurations.
      The function does not return the matrices themselves. It instead
      returns an association whose keys are numE and whose values are
      the filenames where the output of TabulateManyJJBlockMagDipTables
      was saved to. The output consists of an association whose keys are
       of the form {n, J, Jp} and whose values are rectangular arrays
      given the values of <|LSJMJa|H_dip|L'S'J'MJ'a'|>.";
3178    Options[TabulateManyJJBlockMagDipTables]={"FilenameAppendix"->"","
      Overwrite"->False,"Compressed"->True};
3179    TabulateManyJJBlockMagDipTables[ns_,OptionsPattern[]] := (
3180      fnames=<||>;
3181      Do[
3182      (
3183        ExportFun=If[OptionValue["Compressed"],ExportMZip,Export];
3184        PrintTemporary["#------- numE = ",numE," -------#"];
3185        appendTo     = (OptionValue["FilenameAppendix"]<>"-magDip");
3186        exportFname  = JJBlockMatrixFileName[numE,"FilenameAppendix"->
```

```
  appendTo];
3187      fnames[numE] = exportFname;
3188      If[FileExistsQ[exportFname]&&Not[OptionValue["Overwrite"]],
3189        Continue[]
3190      ];
3191      JJBlockMatrixTable = TabulateJJBlockMagDipTable[numE];
3192      If[FileExistsQ[exportFname] && OptionValue["Overwrite"],
3193        DeleteFile[exportFname]
3194      ];
3195      ExportFun[exportFname,  JJBlockMatrixTable];
3196    ),
3197    {numE,ns}
3198    ];
3199    Return[fnames];
3200  );
3201
3202  MagDipoleMatrixAssembly::usage = "MagDipoleMatrixAssembly[numE]
     returns the matrix representation of the operator - 1/2 (L + gs S)
      in the f^numE configuration. The function returns a list with
     three elements corresponding to the x,y,z components of this
     operator. The option \"FilenameAppendix\" can be used to append a
     string to the filename from which the function imports from in
     order to patch together the array. For numE beyond 7 the function
     returns the same as for the complementary configuration. The
     option \"ReturnInBlocks\" can be used to return the matrices in
     blocks. The default is to return the matrices in flattened form
     and as sparse array.";
3203  Options[MagDipoleMatrixAssembly]={
3204    "FilenameAppendix"->"",
3205    "ReturnInBlocks"->False};
3206  MagDipoleMatrixAssembly[nf_Integer, OptionsPattern[]] := Module[
3207    {ImportFun, numE, appendTo,
3208    emFname, JJBlockMagDipTable,
3209    Js, howManyJs, blockOp,
3210    rowIdx, colIdx},
3211    (
3212      ImportFun = ImportMZip;
3213      numE      = nf;
3214      numH      = 14 - numE;
3215      numE      = Min[numE,numH];
3216
3217      appendTo  = (OptionValue["FilenameAppendix"]<>"-magDip");
3218      emFname   = JJBlockMatrixFileName[numE,"FilenameAppendix"->
     appendTo];
3219      JJBlockMagDipTable = ImportFun[emFname];
3220
3221      Js        = AllowedJ[numE];
3222      howManyJs = Length[Js];
3223      blockOp   = ConstantArray[0,{howManyJs,howManyJs}];
3224      Do[
3225        blockOp[[rowIdx,colIdx]] = JJBlockMagDipTable[{numE,Js[[
     rowIdx]],Js[[colIdx]]}],
3226        {rowIdx,1,howManyJs},
3227        {colIdx,1,howManyJs}
3228      ];
3229      If[OptionValue["ReturnInBlocks"],
3230        (
3231          opMinus = Map[#[[1]]&, blockOp, {4}];
3232          opZero  = Map[#[[2]]&, blockOp, {4}];
3233          opPlus  = Map[#[[3]]&, blockOp, {4}];
3234          opX =   (opMinus - opPlus)/Sqrt[2];
3235          opY = I (opPlus + opMinus)/Sqrt[2];
3236          opZ = opZero;
3237        ),
3238          blockOp = ArrayFlatten[blockOp];
3239          opMinus = blockOp[[;; , ;; , 1]];
3240          opZero  = blockOp[[;; , ;; , 2]];
3241          opPlus  = blockOp[[;; , ;; , 3]];
3242          opX =   (opMinus - opPlus)/Sqrt[2];
3243          opY = I (opPlus + opMinus)/Sqrt[2];
3244          opZ = opZero;
3245      ];
3246      Return[{opX, opY, opZ}];
3247    )
3248  ];
3249
```

```
3250  MagDipLineStrength::usage = "MagDipLineStrength[theEigensys, numE]
        takes the eigensystem of an ion and the number numE of f-electrons
         that correspond to it and calculates the line strength array Stot
        .
3251  The option \"Units\" can be set to either \"SI\" (so that the units
         of the returned array are (A m^2)^2) or to \"Hartree\".
3252  The option \"States\" can be used to limit the states for which the
         line strength is calculated. The default, All, calculates the
        line strength for all states. A second option for this is to
        provide an index labelling a specific state, in which case only
        the line strengths between that state and all the others are
        computed.
3253  The returned array should be interpreted in the eigenbasis of the
        Hamiltonian. As such the element Stot[[i,i]] corresponds to the
        line strength states between states |i> and |j>.";
3254  Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI"
        , "States" -> All};
3255  MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern
        []] := Module[
3256    {numE, allEigenvecs, Sx, Sy, Sz, Stot, factor},
3257    (
3258      numE = Min[14-numE0, numE0];
3259      (*If not loaded then load it, *)
3260      If[Or[
3261          Not[MemberQ[Keys[magOp], numE]],
3262          OptionValue["Reload MagOp"]],
3263      (
3264        magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
        MagDipoleMatrixAssembly[numE];
3265      )
3266      ];
3267      allEigenvecs = Transpose[Last /@ theEigensys];
3268      Which[OptionValue["States"] === All,
3269        (
3270          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
        allEigenvecs) & /@ magOp[numE];
3271          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
3272        ),
3273        IntegerQ[OptionValue["States"]],
3274        (
3275          singleState  = theEigensys[[OptionValue["States"],2]];
3276          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
        singleState) & /@ magOp[numE];
3277          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
3278        )
3279      ];
3280      Which[
3281        OptionValue["Units"] == "SI",
3282          Return[4 \[Mu]B^2 * Stot],
3283        OptionValue["Units"] == "Hartree",
3284          Return[Stot],
3285        True,
3286        (
3287          Echo["Invalid option for \"Units\". Options are \"SI\" and
        \"Hartree\"."];
3288          Abort[];
3289        )
3290      ];
3291    )
3292  ];
3293
3294  MagDipoleRates::usage = "MagDipoleRates[eigenSys, numE] calculates
        the magnetic dipole transition rate array for the provided
        eigensystem. The option \"Units\" can be set to \"SI\" or to \"
        Hartree\". If the option \"Natural Radiative Lifetimes\" is set to
         true then the reciprocal of the rate is returned instead.
        eigenSys is a list of lists with two elements, in each list the
        first element is the energy and the second one the corresponding
        eigenvector.
3295  Based on table 7.3 of Thorne 1999, using g2=1.
3296  The energy unit assumed in eigenSys is kayser.
3297  The returned array should be interpreted in the eigenbasis of the
        Hamiltonian. As such the element AMD[[i,i]] corresponds to the
        transition rate (or the radiative lifetime, depending on options)
        between eigenstates |i> and |j>.
3298  By default this assumes that the refractive index is unity, this
```

```
    may be changed by setting the option \"RefractiveIndex\" to the
    desired value.
3299  The option \"Lifetime\" can be used to return the reciprocal of the
       transition rates. The default is to return the transition rates."
    ;
3300  Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False, "
    RefractiveIndex"->1};
3301  MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]] :=
    Module[
3302    {AMD, Stot, eigenEnergies,
3303    transitionWaveLengthsInMeters, nRefractive},
3304    (
3305      nRefractive   = OptionValue["RefractiveIndex"];
3306      numE          = Min[14-numE0, numE0];
3307      Stot          = MagDipLineStrength[eigenSys, numE, "Units" ->
    OptionValue["Units"]];
3308      eigenEnergies = Chop[First/@eigenSys];
3309      energyDiffs   = Outer[Subtract, eigenEnergies, eigenEnergies];
3310      energyDiffs   = ReplaceDiagonal[energyDiffs, Indeterminate];
3311      (* Energies assumed in kayser.*)
3312      transitionWaveLengthsInMeters = 0.01/energyDiffs;
3313
3314      unitFactor    = Which[
3315      OptionValue["Units"]=="Hartree",
3316      (
3317        (* The bohrRadius factor in SI needed to convert the
    wavelengths which are assumed in m*)
3318        16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckHartree)) * bohrRadius^3
3319      ),
3320      OptionValue["Units"]=="SI",
3321      (
3322        16 \[Pi]^3 \[Mu]0/(3 hPlanck)
3323      ),
3324      True,
3325      (
3326        Echo["Invalid option for \"Units\". Options are \"SI\" and \"
    Hartree\"."];
3327        Abort[];
3328      )
3329      ];
3330      AMD = unitFactor / transitionWaveLengthsInMeters^3 * Stot *
    nRefractive^3;
3331      Which[OptionValue["Lifetime"],
3332        Return[1/AMD],
3333        True,
3334        Return[AMD]
3335      ]
3336    )
3337  ];
3338
3339  GroundMagDipoleOscillatorStrength::usage = "
    GroundMagDipoleOscillatorStrength[eigenSys, numE] calculates the
    magnetic dipole oscillator strengths between the ground state and
    the excited states as given by eigenSys.
3340  Based on equation 8 of Carnall 1965, removing the 2J+1 factor since
       this degeneracy has been removed by the crystal field.
3341  eigenSys is a list of lists with two elements, in each list the
    first element is the energy and the second one the corresponding
    eigenvector.
3342  The energy unit assumed in eigenSys is Kayser.
3343  The oscillator strengths are dimensionless.
3344  The returned array should be interpreted in the eigenbasis of the
    Hamiltonian. As such the element fMDGS[[i]] corresponds to the
    oscillator strength between ground state and eigenstate |i>.
3345  By default this assumes that the refractive index is unity, this
    may be changed by setting the option \"RefractiveIndex\" to the
    desired value.";
3346  Options[GroundMagDipoleOscillatorStrength]={"RefractiveIndex"->1};
3347  GroundMagDipoleOscillatorStrength[eigenSys_List, numE_Integer,
    OptionsPattern[]] := Module[
3348    {eigenEnergies, SMDGS, GSEnergy, energyDiffs,
3349    transitionWaveLengthsInMeters, unitFactor, nRefractive},
3350    (
3351      eigenEnergies    = First/@eigenSys;
3352      nRefractive      = OptionValue["RefractiveIndex"];
3353      SMDGS            = MagDipLineStrength[eigenSys,numE, "Units"->"
```

```
    SI", "States"->1];
3354      GSEnergy         = eigenSys[[1,1]];
3355      energyDiffs      = eigenEnergies-GSEnergy;
3356      energyDiffs[[1]] = Indeterminate;
3357      transitionWaveLengthsInMeters = 0.01/energyDiffs;
3358      unitFactor       = (8\[Pi]^2 me)/(3 hPlanck eCharge^2 cLight);
3359      fMDGS            = unitFactor / transitionWaveLengthsInMeters *
     SMDGS * nRefractive;
3360      Return[fMDGS];
3361    )
3362  ];
3363
3364  UkOperator::usage = "UkOperator[numE, k] provides the matrix
      representation of the symmetric unit tensor operator U^k in the
      LSJMJ basis. The function returns an association with keys of the
      form {k,q} where q ranges from -k to k and  values representing
      the different components of the tensor operator. This is computed
      by using the doubly reduced matrix elements provided by
      ReducedUkTable with adequate coupling coefficients.";
3365  UkOperator[numE0_, k0_] := Module[
3366    {
3367      k = k0,
3368      basis,
3369      numE = numE0,
3370      opComponents,
3371      basisSize,
3372      val,
3373      coupling1,
3374      coupling2,
3375      phase,
3376      braLS,
3377      braJ,
3378      braM,
3379      ketLS,
3380      ketJ,
3381      ketM,
3382      braS,
3383      braL,
3384      ketS,
3385      ketL,
3386      q
3387      },
3388    (
3389      If[Not@ValueQ[ReducedUkTable],
3390      LoadUk[]
3391      ];
3392      basis = BasisLSJMJ[numE];
3393      basisSize = Length[basis];
3394      opComponents = <||>;
3395      numE = Min[14-numE, numE];
3396      Do[
3397      (
3398        nonZ = Reap[
3399          Do[(
3400            {braLS, braJ, braM} = basis[[rowIdx]];
3401            {ketLS, ketJ, ketM} = basis[[colIdx]];
3402
3403            redUk = ReducedUkTable[{numE, 3, braLS, ketLS, k}];
3404            If[redUk == 0, Continue[]];
3405
3406            coupling1 = ThreeJay[{ketJ, -ketM}, {k, q}, {braJ, braM
     }];
3407            If[coupling1 == 0, Continue[]];
3408
3409            {braS, braL} = FindSL[braLS];
3410            {ketS, ketL} = FindSL[ketLS];
3411
3412            coupling2 = Sqrt[TPO[braJ]*TPO[ketJ]] * SixJay[{ketL,
     ketJ, ketS}, {braJ, braL, k}];
3413            If[coupling2 == 0, Continue[]];
3414
3415            phase = Phaser[braS + k + braJ + ketL + braJ - braM];
3416            val = N[phase * coupling1 * coupling2 * redUk];
3417            Sow[({rowIdx, colIdx} -> val)]
3418            ),
3419            {rowIdx, 1, basisSize},
```

```
            {colIdx, 1, basisSize}]
          ][[2, 1]];
        opComponents[{k, q}] = SparseArray[nonZ, {basisSize,
  basisSize}];
        ),
      {q, -k, k}];
      Return[opComponents]
      )
    ];

EffectiveElectricDipole::usage = "EffectiveElectricDipole[numE,
  Aparams] calculates the effective dipole operator in configuration
   f^numE for the given intensity parameters Aparams (given as an
  Association). The function returns an Association with three keys
  {-1,0,1} representing the three components of the operator and
  with values equal to arrays for the corresponding matrix
  representations in the standard LSJMJ basis. The units being equal
   to those of the units assumed for Aparams.";
EffectiveElectricDipole[numE0_, Aparams0_] := Module[
  {
    Aparams = Aparams0,
    numE = numE0,
    Deff,
    keys,
    key,
    altKey,
    Aparam,
    qComponents
    },
  (
    keys = Keys[Aparams];
    Deff = <||>;
    qComponents = {-1, 0, 1};
    If[Not[ValueQ[Uks]],
      Uks = <||>
    ];
    Do[
    (
      sumTerms = Reap[
        Do[
          (
          key    = {\[Lambda], t, p};
          altKey = {\[Lambda], t, -p};
          (* the way the iterator is carried out, some CG are non-
  physical, leave them out early *)
          If[
            Or[Abs[p] > t,
            Abs[p + q] > \[Lambda]
            ],
          Continue[]
          ];
          (* enforce relation between conjugate values *)
          Aparam = Which[
            MemberQ[keys, key],
            Aparams[key],
            MemberQ[keys, altKey],
            Conjugate[Phaser[t + p + 1] * Aparams[altKey]],
            True,
            Continue[]
            ];
          (* calculate the Uk operator if it has not been
  calculated *)
          If[Not@MemberQ[Keys[Uks], \[Lambda]],
            Uks[\[Lambda]] = UkOperator[numE, \[Lambda]];
            ];

          clebscG = ClebschGordan[
            {\[Lambda], p + q},
            {1, -q},
            {t, p}];
          If[clebscG === 0, Continue[]];

          Sow[Phaser[q]*Aparam*clebscG*Uks[\[Lambda]][{\[Lambda], p
   + q}]];
          ),
          {\[Lambda], {2, 4, 6}},
```

```
            {t, {\[Lambda] - 1, \[Lambda], \[Lambda] + 1}},
            {p, -t, t, 1}
            ]
          ][[2, 1]];
        Deff[q] = Total[N@sumTerms];
        ),
      {q, qComponents}
      ];
      Return[Deff]
      )
    ];

ElectricDipLineStrength::usage = "ElectricDipLineStrength[
  theEigensys, numE, Aparams] takes the eigensystem of an ion with
  configuration f^numE together with the intensity parameters A that
   define the effective electric dipole operator.
The option \"Units\" can be set to either \"SI\" (so that the units
   of the returned array are (units of A)^2 * C^2) or to \"Hartree-
  partial\" in which case the units of the returned array are
  determined by the units of Aparams and equal to (units of Aparams)
  ^2.
The option \"States\" can be used to limit the states for which the
   line strength is calculated. The default, All, calculates the
  line strength for all states. A second option for this is to
  provide an index labelling a specific state, in which case only
  the line strengths between that state and all the others are
  computed.
The returned array should be interpreted in the eigenbasis of the
  Hamiltonian. As such the element Sed[[i,i]] corresponds to the
  line strength states between states |i> and |j>.
In what is returned no sum is made over degenerate states.
";
Options[ElectricDipLineStrength] = {"Units" -> "SI","States" -> All
  };
ElectricDipLineStrength[theEigensys_List, numE0_Integer,
  Aparams_Association, OptionsPattern[]] := Module[
    {
      numE = numE0,
      Deffcalc,
      allEigenvecs,
      Sed
    },
    (
      numE = Min[14-numE, numE];
      Deff = EffectiveElectricDipole[numE, Aparams];
      allEigenvecs = Transpose[Last /@ theEigensys];

      Which[OptionValue["States"] === All,
        (
          Deffcalc = (ConjugateTranspose[allEigenvecs] . # .
  allEigenvecs) & /@ Deff;
        ),
        IntegerQ[OptionValue["States"]],
        (
          singleState  = theEigensys[[OptionValue["States"],2]];
          Deffcalc = (ConjugateTranspose[allEigenvecs] . # .
  singleState) & /@ Deff;
        )
      ];
      Sed  = (Abs[Deffcalc[-1]^2] + Abs[Deffcalc[0]^2] + Abs[Deffcalc
  [1]^2]);
      Which[
        OptionValue["Units"] == "SI",
        Return[eCharge^2 * Sed],
        OptionValue["Units"] == "Hartree-partial",
        Return[Sed]
      ];
    )
];

(* ##################### Optical Operators ##################### *)
(* ############################################################ *)

(* ############################################################ *)
(* ################### Printers and Labels ################### *)

```

```
PrintL::usage = "PrintL[L] give the string representation of a
  given angular momentum.";
PrintL[L_] := If[StringQ[L], L, StringTake[specAlphabet, {L + 1}]]

FindSL::usage = "FindSL[LS] gives the spin and orbital angular
  momentum that corresponds to the provided string LS.";
FindSL[SL_] := (
  FindSL[SL] =
  If[StringQ[SL],
    {
      (ToExpression[StringTake[SL, 1]]-1)/2,
      StringPosition[specAlphabet, StringTake[SL, {2}]][[1, 1]]-1
    },
    SL
  ]
);

PrintSLJ::usage = "Given a list with three elements {S, L, J} this
  function returns a symbol where the spin multiplicity is presented
   as a superscript, the orbital angular momentum as its
  corresponding spectroscopic letter, and J as a subscript. Function
   does not check to see if the given J is compatible with the given
   S and L.";
PrintSLJ[SLJ_] := (
  RowBox[{
    SuperscriptBox[" ", 2 SLJ[[1]] + 1],
    SubscriptBox[PrintL[SLJ[[2]]], SLJ[[3]]]
    }
    ] // DisplayForm
  );

PrintSLJM::usage = "Given a list with four elements {S, L, J, MJ}
  this function returns a symbol where the spin multiplicity is
  presented as a superscript, the orbital angular momentum as its
  corresponding spectroscopic letter, and {J, MJ} as a subscript. No
   attempt is made to guarantee that the given input is consistent."
  ;
PrintSLJM[SLJM_] := (
  RowBox[{
    SuperscriptBox[" ", 2 SLJM[[1]] + 1],
    SubscriptBox[PrintL[SLJM[[2]]], {SLJM[[3]], SLJM[[4]]}]
    }
    ] // DisplayForm
  );

(* ################### Printers and Labels ################### *)
(* ############################################################ *)

(* ############################################################ *)
(* ###################### Term management ###################### *)

AllowedSLTerms::usage = "AllowedSLTerms[numE] returns a list with
  the allowed terms in the f^numE configuration, the terms are given
   as lists in the format {S, L}. This list may have redundancies
  which are compatible with the degeneracies that might correspond
  to the given case.";
AllowedSLTerms[numE_] := Map[FindSL[First[#]] &, CFPTerms[Min[numE,
   14-numE]]];

AllowedNKSLTerms::usage = "AllowedNKSLTerms[numE] returns a list
  with the allowed terms in the f^numE configuration, the terms are
  given as strings in spectroscopic notation. The integers in the
  last positions are used to distinguish cases with degeneracy.";
AllowedNKSLTerms[numE_] := Map[First, CFPTerms[Min[numE, 14-numE
  ]]];
AllowedNKSLTerms[0]  = {"1S"};
AllowedNKSLTerms[14] = {"1S"};

MaxJ::usage = "MaxJ[numE] gives the maximum J = S+L that
  corresponds to the configuration f^numE.";
MaxJ[numE_] := Max[Map[Total, AllowedSLTerms[Min[numE, 14-numE]]]];

MinJ::usage = "MinJ[numE] gives the minimum J = S+L that
  corresponds to the configuration f^numE.";
MinJ[numE_] := Min[Map[Abs[Part[#, 1] - Part[#, 2]] &,
  AllowedSLTerms[Min[numE, 14-numE]]]]
```

```
3594
3595  AllowedSLJTerms::usage = "AllowedSLJTerms[numE] returns a list with
        the allowed {S, L, J} terms in the f^n configuration, the terms
       are given as lists in the format {S, L, J}. This list may have
       repeated elements which account for possible degeneracies of the
       related term.";
3596  AllowedSLJTerms[numE_] := Module[
3597    {idx1, allowedSL, allowedSLJ},
3598    (
3599      allowedSL = AllowedSLTerms[numE];
3600      allowedSLJ = {};
3601      For[
3602        idx1 = 1,
3603        idx1 <= Length[allowedSL],
3604        termSL = allowedSL[[idx1]];
3605        termsSLJ =
3606          Table[
3607            {termSL[[1]], termSL[[2]], J},
3608          {J,Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
3609          ];
3610        allowedSLJ = Join[allowedSLJ, termsSLJ];
3611        idx1++
3612      ];
3613      SortBy[allowedSLJ, Last]
3614    )
3615  ];
3616
3617  AllowedNKSLJTerms::usage = "AllowedNKSLJTerms[numE] returns a list
       with the allowed {SL, J} terms in the f^n configuration, the terms
        are given as lists in the format {SL, J} where SL is a string in
       spectroscopic notation.";
3618  AllowedNKSLJTerms[numE_] := Module[
3619    {allowedSL, allowedNKSL, allowedSLJ, nn},
3620    (
3621      allowedNKSL = AllowedNKSLTerms[numE];
3622      allowedSL = AllowedSLTerms[numE];
3623      allowedSLJ = {};
3624      For[
3625        nn = 1,
3626        nn <= Length[allowedSL],
3627        (
3628          termSL = allowedSL[[nn]];
3629          termNKSL = allowedNKSL[[nn]];
3630          termsSLJ =
3631            Table[{termNKSL, J},
3632            {J, Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
3633            ];
3634          allowedSLJ = Join[allowedSLJ, termsSLJ];
3635          nn++
3636        )
3637      ];
3638      SortBy[allowedSLJ, Last]
3639    )
3640  ];
3641
3642  AllowedNKSLforJTerms::usage = "AllowedNKSLforJTerms[numE, J] gives
       the terms that correspond to the given total angular momentum J in
        the f^n configuration. The result is a list whose elements are
       lists of length 2, the first element being the SL term in
       spectroscopic notation, and the second element being J.";
3643  AllowedNKSLforJTerms[numE_, J_] := Module[
3644    {allowedSL, allowedNKSL, allowedSLJ,
3645    nn, termSL, termNKSL, termsSLJ},
3646    (
3647      allowedNKSL = AllowedNKSLTerms[numE];
3648      allowedSL = AllowedSLTerms[numE];
3649      allowedSLJ = {};
3650      For[
3651        nn = 1,
3652        nn <= Length[allowedSL],
3653        (
3654          termSL = allowedSL[[nn]];
3655          termNKSL = allowedNKSL[[nn]];
3656          termsSLJ = If[Abs[termSL[[1]] - termSL[[2]]] <= J <= Total[
    termSL],
3657            {{termNKSL, J}},
```

```
            {}
            ];
          allowedSLJ = Join[allowedSLJ, termsSLJ];
          nn++
        )
      ];
      Return[allowedSLJ]
    )
  ];

  AllowedSLJMTerms::usage = "AllowedSLJMTerms[numE] returns a list
    with all the states that correspond to the configuration f^n. A
    list is returned whose elements are lists of the form {S, L, J, MJ
    }.";
  AllowedSLJMTerms[numE_] := Module[
    {allowedSLJ, allowedSLJM,
    termSLJ, termsSLJM, nn},
    (
    allowedSLJ = AllowedSLJTerms[numE];
    allowedSLJM = {};
    For[
      nn = 1,
      nn <= Length[allowedSLJ],
      nn++,
      (
        termSLJ = allowedSLJ[[nn]];
        termsSLJM =
          Table[{termSLJ[[1]], termSLJ[[2]], termSLJ[[3]], M},
          {M, - termSLJ[[3]], termSLJ[[3]]}
          ];
        allowedSLJM = Join[allowedSLJM, termsSLJM];
      )
    ];
    Return[SortBy[allowedSLJM, Last]];
    )
  ];

  AllowedNKSLJMforJMTerms::usage = "AllowedNKSLJMforJMTerms[numE, J,
    MJ] returns a list with all the terms that contain states of the f
    ^n configuration that have a total angular momentum J, and a
    projection along the z-axis MJ. The returned list has elements of
    the form {SL (string in spectroscopic notation), J, MJ}.";
  AllowedNKSLJMforJMTerms[numE_, J_, MJ_] := Module[
    {allowedSL, allowedNKSL,
    allowedSLJM, nn},
    (
      allowedNKSL = AllowedNKSLTerms[numE];
      allowedSL   = AllowedSLTerms[numE];
      allowedSLJM = {};
      For[
        nn = 1,
        nn <= Length[allowedSL],
        termSL = allowedSL[[nn]];
        termNKSL = allowedNKSL[[nn]];
        termsSLJ = If[(Abs[termSL[[1]] - termSL[[2]]]
                      <= J
                      <= Total[termSL]
                      && (Abs[MJ] <= J)
                      ),
                      {{termNKSL, J, MJ}},
                      {}];
        allowedSLJM = Join[allowedSLJM, termsSLJ];
        nn++
      ];
      Return[allowedSLJM];
    )
  ];

  AllowedNKSLJMforJTerms::usage = "AllowedNKSLJMforJTerms[numE, J]
    returns a list with all the states that have a total angular
    momentum J. The returned list has elements of the form {SL (string
     in spectroscopic notation), J, MJ}.";
  AllowedNKSLJMforJTerms[numE_, J_] := Module[
    {MJs, labelsAndMomenta, termsWithJ},
    (
      MJs = AllowedMforJ[J];
```

```
3724        (* Pair LS labels and their {S,L} momenta *)
3725        labelsAndMomenta = ({#, FindSL[#]}) & /@ AllowedNKSLTerms[numE
    ];
3726        (* A given term will contain J if |L-S|<=J<=L+S *)
3727        ContainsJ[{SL_String, {S_, L_}}] := (Abs[S - L] <= J <= (S + L)
    );
3728        (* Keep just the terms that satisfy this condition *)
3729        termsWithJ = Select[labelsAndMomenta, ContainsJ];
3730        (* We don't want to keep the {S,L} *)
3731        termsWithJ = {#[[1]], J} & /@ termsWithJ;
3732        (* This is just a quick way of including up all the MJ values
    *)
3733        Return[Flatten /@ Tuples[{termsWithJ, MJs}]]
3734      )
3735    ];
3736
3737    AllowedMforJ::usage = "AllowedMforJ[J] is shorthand for Range[-J, J
      , 1].";
3738    AllowedMforJ[J_] := Range[-J, J, 1];
3739
3740    AllowedJ::usage = "AllowedJ[numE] returns the total angular momenta
       J that appear in the f^numE configuration.";
3741    AllowedJ[numE_] := Table[J, {J, MinJ[numE], MaxJ[numE]}];
3742
3743    Seniority::usage = "Seniority[LS] returns the seniority of the
      given term.";
3744    Seniority[LS_] := FindNKLSTerm[LS][[1, 2]];
3745
3746    FindNKLSTerm::usage = "Given the string LS FindNKLSTerm[SL] returns
       all the terms that are compatible with it. This is only for f^n
      configurations. The provided terms might belong to more than one
      configuration. The function returns a list with elements of the
      form {LS, seniority, W, U}.";
3747    FindNKLSTerm[SL_] := Module[
3748      {NKterms, n},
3749      (
3750        n = 7;
3751        NKterms = {{}};
3752        Map[
3753          If[! StringFreeQ[First[#], SL],
3754            If[ToExpression[Part[#, 2]] <= n,
3755              NKterms = Join[NKterms, {#}, 1]
3756            ]
3757          ] &,
3758        fnTermLabels
3759        ];
3760        NKterms = DeleteCases[NKterms, {}];
3761        NKterms
3762      )
3763    ];
3764
3765    ParseTermLabels::usage = "ParseTermLabels[] parses the labels for
      the terms in the f^n configurations based on the labels for the f6
       and f7 configurations. The function returns a list whose elements
       are of the form {LS, seniority, W, U}.";
3766    Options[ParseTermLabels] = {"Export" -> True};
3767    ParseTermLabels[OptionsPattern[]] := Module[
3768      {labelsTextData, fNtextLabels, nielsonKosterLabels,
3769      seniorities, RacahW, RacahU},
3770      (
3771        labelsTextData = FileNameJoin[{moduleDir, "data", "
      NielsonKosterLabels_f6_f7.txt"}];
3772        fNtextLabels   = Import[labelsTextData];
3773        nielsonKosterLabels = Partition[StringSplit[fNtextLabels], 3];
3774        termLabels = Map[Part[#, {1}] &, nielsonKosterLabels];
3775        seniorities = Map[ToExpression[Part[# , {2}]] &,
      nielsonKosterLabels];
3776        racahW =
3777          Map[
3778            StringTake[
3779              Flatten[StringCases[Part[# , {3}],
3780                "(" ~~ DigitCharacter ~~ DigitCharacter ~~
      DigitCharacter ~~ ")"]],
3781              {2, 4}
3782            ] &,
3783          nielsonKosterLabels];
```

```
3784      racahU =
3785        Map[
3786          StringTake[
3787            Flatten[StringCases[Part[# , {3}],
3788              "(" ~~ DigitCharacter ~~ DigitCharacter ~~ ")"]],
3789            {2, 3}
3790          ] &,
3791        nielsonKosterLabels];
3792      fnTermLabels = Join[termLabels, seniorities, racahW, racahU,
    2];
3793      fnTermLabels = Sort[fnTermLabels];
3794      If[OptionValue["Export"],
3795        (
3796          broadFname = FileNameJoin[{moduleDir,"data","fnTerms.m"}];
3797          Export[broadFname, fnTermLabels];
3798        )
3799      ];
3800      Return[fnTermLabels];
3801    )
3802  ];
3803
3804  (* ##################### Term management ##################### *)
3805  (* ############################################################ *)
3806
3807  LoadLaF3Parameters::usage="LoadLaF4Parameters[ln] gives the models
      for the semi-empirical Hamiltonian for the trivalent lanthanide
      ion with symbol ln.";
3808  Options[LoadLaF3Parameters] = {
3809    "Vintage"  -> "Standard",
3810    "With Uncertainties"->False
3811  };
3812  LoadLaF3Parameters[Ln_String, OptionsPattern[]]:= Module[
3813    {paramData, params},
3814    (
3815      paramData = Which[
3816        OptionValue["Vintage"] == "Carnall",
3817        Import[FileNameJoin[{moduleDir, "data", "carnallParams.m"}]],
3818        OptionValue["Vintage"] == "Standard",
3819        Import[FileNameJoin[{moduleDir, "data", "standardParams.m"}]]
3820      ];
3821    params = If[OptionValue["With Uncertainties"],
3822      paramData[Ln],
3823      If[Head[#] === Around, #["Value"], #] & /@ paramData[Ln]
3824    ];
3825    Return[params];
3826    )
3827  ];
3828
3829  HoleElectronConjugation::usage = "HoleElectronConjugation[params]
      takes the parameters (as an association) that define a
      configuration and converts them so that they may be interpreted as
       corresponding to a complementary hole configuration. Some of this
       can be simply done by changing the sign of the model parameters.
      In the case of the effective three body interaction of T2 the
      relationship is more complex and is controlled by the value of the
       t2Switch variable.";
3830  HoleElectronConjugation[params_] := Module[
3831    {newparams = params},
3832    (
3833      flipSignsOf = Join[{ζ} , cfSymbols, TSymbols];
3834      flipped = Table[
3835        (
3836          flipper -> - newparams[flipper]
3837        ),
3838        {flipper, flipSignsOf}
3839        ];
3840      nonflipped = Table[
3841        (
3842          flipper -> newparams[flipper]
3843        ),
3844        {flipper, Complement[Keys[newparams], flipSignsOf]}
3845        ];
3846      flippedParams = Association[Join[nonflipped, flipped]];
3847      flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
3848      Return[flippedParams];
3849    )
```

```
3850  ];
3851
3852  IonSolver::usage = "IonSolver[numE, params, host] puts together (or
        retrieves from disk) the symbolic Hamiltonian for the f^numE
      configuration and solves it for the given params.
3853  params is an Association with keys equal to parameter symbols and
      values their numerical values. The function will replace the
      symbols in the symbolic Hamiltonian with their numerical values
      and then diagonalize the resulting matrix. Any parameter that is
      not defined in the params Association is assumed to be zero.
3854  host is an optional string that may be used to prepend the filename
        of the symbolic Hamiltonian that is saved to disk. The default is
        \"Ln\".
3855  The function returns the eigensystem as a list of lists where in
      each list the first element is the energy and the second element
      the corresponding eigenvector.
3856  The ordered basis in which these eigenvectors are to be interpreted
        is the one corresponding to BasisLSJMJ[numE].
3857  The function admits the following options:
3858  \"Include Spin-Spin\" (bool) : If True then the spin-spin
      interaction is included as a contribution to the m_k operators.
      The default is True.
3859  \"Overwrite Hamiltonian\" (bool) : If True then the function will
      overwrite the symbolic Hamiltonian that is saved to disk to
      expedite calculations. The default is False. The symbolic
      Hamiltonian is saved to disk to the ./hams/ folder preceded by the
       string host.
3860  \"Zeroes\" (list) : A list with symbols assumed to be zero.
3861  ";
3862  Options[IonSolver] = {
3863    "Include Spin-Spin" -> True,
3864    "Overwrite Hamiltonian" -> False,
3865    "Zeroes" -> {}
3866  };
3867  IonSolver[numE_Integer, params0_Association, host_String:"Ln",
      OptionsPattern[]] := Module[
3868    {ln, simplifier, simpleHam, numHam, eigensys,
3869    startTime, endTime, diagonalTime,
3870    params=params0, zeroSymbols},
3871    (
3872      ln = theLanthanides[[numE]];
3873
3874      (* This could be done when replacing values, but this produces
      smaller saved arrays. *)
3875      simplifier = (#-> 0) & /@ OptionValue["Zeroes"];
3876      simpleHam = SimplerEffectiveHamiltonian[numE,
3877        simplifier,
3878        "PrependToFilename" -> host,
3879        "Overwrite" -> OptionValue["Overwrite Hamiltonian"]
3880      ];
3881
3882      (* Note that we don't have to flip signs of parameters for fn
      beyond f7 since the matrix produced
3883      by SimplerEffectiveHamiltonian has already accounted for this.
      *)
3884
3885      (* Everything that is not given is set to zero *)
3886      params = ParamPad[params];
3887      PrintFun[params];
3888
3889      (* Enforce the override to the spin-spin contribution to the
      magnetic interactions *)
3890      params[\[Sigma]SS] = If[OptionValue["Include Spin-Spin"], 1,
      0];
3891
3892      (* Create the numeric hamiltonian *)
3893      numHam = ReplaceInSparseArray[simpleHam, params];
3894      Clear[simpleHam];
3895
3896      (* Eigensolver *)
3897      PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
3898      startTime = Now;
3899      eigensys  = Eigensystem[numHam];
3900      endTime   = Now;
3901      diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"
      ];
```

```
3902        PrintFun[">> Diagonalization took ", diagonalTime, " seconds."
      ];
3903        eigensys = Chop[eigensys];
3904        eigensys = Transpose[eigensys];
3905
3906        (* Shift the baseline energy *)
3907        eigensys = ShiftedLevels[eigensys];
3908        (* Sort according to energy *)
3909        eigensys = SortBy[eigensys, First];
3910        Return[eigensys];
3911      )
3912    ];
3913
3914    ShiftedLevels::usage = "ShiftedLevels[eigenSys] takes a list of
        levels of the form
3915    {{energy_1, coeff_vector_1}, {energy_2, coeff_vector_2},...}} and
        returns the same input except that now to every energy the minimum
         of all of them has been subtracted.";
3916    ShiftedLevels[originalLevels_] := Module[
3917      {groundEnergy, shifted},
3918      (
3919        groundEnergy = Sort[originalLevels][[1,1]];
3920        shifted      = Map[{#[[1]] - groundEnergy, #[[2]]} &,
      originalLevels];
3921        Return[shifted];
3922      )
3923    ];
3924
3925
3926    (* ############################################################ *)
3927    (* ############# Optical Transitions for Levels ############## *)
3928
3929    JuddOfeltUkSquared::usage = "JuddOfeltUkSquared[numE, params]
        calculates the matrix elements of the Uk operator in the level
        basis. These are calculated according to equation (7) in Carnall
        1965.
3930    The function returns a list with the following elements:
3931      - basis : A list with the allowed {SL, J} terms in the f^numE
        configuration. Equal to BasisLSJ[numE].
3932      - eigenSys : A list with the eigensystem of the Hamiltonian for
        the f^n configuration.
3933      - levelLabels : A list with the labels of the major components of
         the level eigenstates.
3934      - LevelUkSquared : An association with the squared matrix
        elements of the Uk operators in the level eigenbasis. The keys
        being {2, 4, 6} corresponding to the rank of the Uk operator. The
        basis in which the matrix elements are given is the one
        corresponding to the level eigenstates given in eigenSys and whose
         major SLJ components are given in levelLabels. The matrix is
        symmetric and given as a SymmetrizedArray.
3935      The function admits the following options:
3936      \"PrintFun\" : A function that will be used to print the progress
         of the calculations. The default is PrintTemporary.";
3937    Options[JuddOfeltUkSquared] = {"PrintFun" -> PrintTemporary};
3938    JuddOfeltUkSquared[numE_, params_, OptionsPattern[]] := Module[
3939      {eigenChanger, numEH, basis, eigenSys,
3940      Js, Ukmat, LevelUkSquared, kRank,
3941      S, L, Sp, Lp, J, Jp, phase,
3942      braTerm, ketTerm, levelLabels,
3943      eigenVecs, majorComponentIndices},
3944      (
3945        If[Not[ValueQ[ReducedUkTable]],
3946          LoadUk[]
3947        ];
3948        numEH = Min[numE, 14-numE];
3949        PrintFun = OptionValue["PrintFun"];
3950        PrintFun["> Calculating the levels for the given parameters ...
      "];
3951        {basis, majorComponents, eigenSys} = LevelSolver[numE, params];
3952        (* The change of basis matrix to the eigenstate basis *)
3953        eigenChanger = Transpose[Last /@ eigenSys];
3954        PrintFun["Calculating the matrix elements of Uk in the physical
         coupling basis ..."];
3955        LevelUkSquared = <||>;
3956        Do[(
3957          Ukmat = Table[(
```

```
            {S, L}   = FindSL[braTerm[[1]]];
            J        = braTerm[[2]];
            Jp       = ketTerm[[2]];
            {Sp, Lp} = FindSL[ketTerm[[1]]];
            phase    = Phaser[S + Lp + J + kRank];
            Simplify @ (
              phase *
              Sqrt[TPO[J]*TPO[Jp]] *
              SixJay[{J, Jp, kRank}, {Lp, L, S}] *
              ReducedUkTable[{numEH, 3, braTerm[[1]], ketTerm[[1]], kRank}]
              )
            ),
          {braTerm, basis},
          {ketTerm, basis}
        ];
        Ukmat = (Transpose[eigenChanger] . Ukmat . eigenChanger)^2;
        Ukmat = Chop@Ukmat;
        LevelUkSquared[kRank] = SymmetrizedArray[Ukmat, Dimensions[eigenChanger], Symmetric[{1, 2}]];
        ),
        {kRank, {2, 4, 6}}
      ];
      LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[InputForm[#[[2]]]]) & /@ basis;
      eigenVecs = Last /@ eigenSys;
      majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ eigenVecs;
      levelLabels = LSJmultiplets[[majorComponentIndices]];
      Return[{basis, eigenSys, levelLabels, LevelUkSquared}];
    )
  ];

  LevelElecDipoleOscillatorStrength::usage = "
    LevelElecDipoleOscillatorStrength[numE, levelParams, juddOfeltParams] uses Judd-Ofelt theory to estimate the forced electric dipole oscillator strengths ions whose level description is determined by levelParams.
  The third parameter juddOfeltParams is an association with keys equal to the three Judd-Ofelt intensity parameters {\[CapitalOmega]2, \[CapitalOmega]4, \[CapitalOmega]6} and corresponding values in cm^2.
  The local field correction implemented here corresponds to the one given by the virtual cavity model of Lorentz.
  The function returns a list with the following elements:
    - basis : A list with the allowed {SL, J} terms in the f^numE configuration. Equal to BasisLSJ[numE].
    - eigenSys : A list with the eigensystem of the Hamiltonian for the f^n configuration in the level description.
    - levelLabels : A list with the labels of the major components of the calculated levels.
    - oStrengthArray : A square array whose elements represent the oscillator strengths between levels such that the element oStrengthArray[[i,j]] is the oscillator strength between the levels |Subscript[\[Psi], i]> and |Subscript[\[Psi], j]>. In this array, the elements below the diagonal represent emission oscillator strengths, and elements above the diagonal represent absorption oscillator strengths.
  The function admits the following three options:
    \"PrintFun\" : A function that will be used to print the progress of the calculations. The default is PrintTemporary.
    \"RefractiveIndex\" : The refractive index of the medium where the transitions are taking place. This may be a number or a function. If a number then the oscillator strengths are calculated for assuming a wavelength-independent refractive index. If a function then the refractive indices are calculated accordingly to the wavelength of each transition (the function must admit a single argument equal to the wavelength in nm). The default is 1.
    \"LocalFieldCorrection\" : The local field correction to be used. The default is \"VirtualCavity\". The options are: \"VirtualCavity\" and \"EmptyCavity\".
  The equation implemented here is the one given in eqn. 29 from the review article of Hehlen (2013). See that same article for a discussion on the local field correction.
  ";
  Options[LevelElecDipoleOscillatorStrength]={
    "PrintFun"        -> PrintTemporary,
```

```
4003      "RefractiveIndex" -> 1,
4004      "LocalFieldCorrection" -> "VirtualCavity"
4005    };
4006    LevelElecDipoleOscillatorStrength[numE_, levelParams_Association,
          juddOfeltParams_Association, OptionsPattern[]] := Module[
4007      {PrintFun, basis, eigenSys, levelLabels,
4008      LevelUkSquared, eigenEnergies, energyDiffs,
4009      oStrengthArray, nRef, \[Chi], nRefs,
4010      \[Chi]OverN, groundLevel, const,
4011      transitionFrequencies, wavelengthsInNM,
4012      fieldCorrectionType},
4013      (
4014        PrintFun = OptionValue["PrintFun"];
4015        nRef     = OptionValue["RefractiveIndex"];
4016        PrintFun["Calculating the Uk^2 matrix elements for the given
          parameters ..."];
4017        {basis, eigenSys, levelLabels, LevelUkSquared} =
          JuddOfeltUkSquared[numE, levelParams, "PrintFun"->PrintFun];
4018        eigenEnergies = First/@eigenSys;
4019        (* converted to cm^-2 *)
4020        const         = (8\[Pi]^2)/3 me/hPlanck * 10^(-4);
4021        energyDiffs   = Transpose@Outer[Subtract,eigenEnergies,
          eigenEnergies];
4022        (* since energies are assumed in Kayser, speed of light needs
          to be in cm/s, so that the frequencies are in 1/s *)
4023        transitionFrequencies = energyDiffs*cLight*100;
4024        (* grab the J for each level *)
4025        levelJs        = #[[2]] & /@ eigenSys;
4026        oStrengthArray = (
4027          juddOfeltParams[\[CapitalOmega]2]*LevelUkSquared[2]+
4028          juddOfeltParams[\[CapitalOmega]4]*LevelUkSquared[4]+
4029          juddOfeltParams[\[CapitalOmega]6]*LevelUkSquared[6]
4030        );
4031        oStrengthArray = Abs@(const * transitionFrequencies *
          oStrengthArray);
4032        (* it is necessary to divide each oscillator strength by the
          degeneracy of the initial level *)
4033        oStrengthArray = MapIndexed[1/(2 levelJs[[#2[[1]]]]+1) #1 &,
          oStrengthArray,{2}];
4034        (* including the effects of the refractive index *)
4035        fieldCorrectionType = OptionValue["LocalFieldCorrection"];
4036        Which[
4037          nRef === 1,
4038            True,
4039          NumberQ[nRef],
4040          (
4041            \[Chi] = Which[
4042              fieldCorrectionType == "VirtualCavity",
4043              (
4044                ( (nRef^2 + 2) / 3 )^2
4045              ),
4046              fieldCorrectionType == "EmptyCavity",
4047              (
4048                ( 3 * nRef^2 / ( 2 * nRef^2 + 1 ) )^2
4049              )
4050            ];
4051            \[Chi]OverN    = \[Chi] / nRef;
4052            oStrengthArray = \[Chi]OverN * oStrengthArray;
4053            (* the refractive index participates differently in
          absorption and in emission *)
4054            aFunction      = If[#2[[1]]>#2[[2]], #1 * nRef^2, #1]&;
4055            oStrengthArray = MapIndexed[aFunction, oStrengthArray,
          {2}];
4056          ),
4057          True,
4058          (
4059            wavelengthsInNM = Abs[1 / energyDiffs] * 10^7;
4060            nRefs           = Map[nRef, wavelengthsInNM];
4061            Echo["Calculating the oscillator strengths for the given
          refractive index ..."];
4062            \[Chi] = Which[
4063              fieldCorrectionType == "VirtualCavity",
4064              (
4065                ( (nRefs^2 + 2) / 3 )^2
4066              ),
4067              fieldCorrectionType == "EmptyCavity",
```

```
            (
              ( 3 * nRefs^2 / ( 2*nRefs^2 + 1 ) )^2
            )
          ];
          \[Chi]OverN     = \[Chi] / nRefs;
          oStrengthArray  = \[Chi]OverN * oStrengthArray
        )
      ];
      Return[{basis, eigenSys, levelLabels, oStrengthArray}];
    )
  ];

  LevelJJBlockMagDipole::usage = "LevelJJBlockMagDipole[numE, J, Jp]
    returns an array of the LSJ reduced matrix elements of the
    magnetic dipole operator between states with given J and Jp. The
    option \"Sparse\" can be used to return a sparse matrix. The
    default is to return a sparse matrix.";
  Options[LevelJJBlockMagDipole] = {"Sparse"->True};
  LevelJJBlockMagDipole[numE_, braJ_, ketJ_, OptionsPattern[]] :=
    Module[
    {
      braSLJs, ketSLJs,
      braSLJ, ketSLJ,
      braSL, ketSL,
      braS, braL,
      ketS, ketL,
      matValue, magMatrix,
      summand1, summand2
    },
  (
    braSLJs     = AllowedNKSLforJTerms[numE,braJ];
    ketSLJs     = AllowedNKSLforJTerms[numE,ketJ];
    magMatrix   = Table[
      (
        braSL       = braSLJ[[1]];
        ketSL       = ketSLJ[[1]];
        {braS,braL} = FindSL[braSL];
        {ketS,ketL} = FindSL[ketSL];
        summand1    = If[Or[braJ!=ketJ,braSL!=ketSL],
          0,
          Sqrt[braJ*(braJ+1)*TPO[braJ]
          ]
        ];
        (*looking at the string includes checking L=L',S=S',and\alpha
    =\alpha'*)
        summand2    = If[braSL!=ketSL,
          0,
          (gs-1)*
          Phaser[braS+braL+ketJ+1]*
          Sqrt[TPO[braJ]*TPO[ketJ]]*
          SixJay[{braJ,1,ketJ},{braS,braL,braS}]*
          Sqrt[braS(braS+1)TPO[braS]]
        ];
        matValue = summand1 + summand2;
        matValue = -1/2 * matValue;
        matValue
      ),
      {braSLJ,braSLJs},
      {ketSLJ,ketSLJs}
    ];
    If[OptionValue["Sparse"],
      magMatrix = SparseArray[magMatrix]];
    Return[magMatrix];
  )
  ];

  LevelMagDipoleMatrixAssembly::usage = "LevelMagDipoleMatrixAssembly
    [numE] puts together an array with the reduced matrix elements of
    the magnetic dipole operator in the level basis for the f^numE
    configuration. The function admits the two following options:
    \"Flattened\": If True then the returned matrix is flattened. The
     default is True.
    \"gs\": The electronic gyromagnetic ratio. The default is 2.";
  Options[LevelMagDipoleMatrixAssembly] = {
    "Flattened" -> True,
    gs -> 2
```

```
};
LevelMagDipoleMatrixAssembly[numE_, OptionsPattern[]] := Module[
  {Js, magDip, braJ, ketJ},
  (
    Js      = AllowedJ[numE];
    magDip  = Table[
      ReplaceInSparseArray[LevelJJBlockMagDipole[numE, braJ, ketJ],
 {gs -> OptionValue[gs]}],
      {braJ,Js},
      {ketJ,Js}
    ];
    If[OptionValue["Flattened"],
      magDip = ArrayFlatten[magDip];
    ];
    Return[magDip];
  )
];

LevelMagDipoleLineStrength::usage = "LevelMagDipoleLineStrength[
  eigenSys, numE] calculates the magnetic dipole line strengths for
  an ion whose level description is determined by levelParams. The
  function returns a square array whose elements represent the
  magnetic dipole line strengths between the levels given in
  eigenSys such that the element magDipoleLineStrength[[i,j]] is the
   line strength between the levels |Subscript[\[Psi], i]> and |
  Subscript[\[Psi], j]>. Eigensys must be such that it consists of a
   lists of lists where in each list the last element corresponds to
   the eigenvector of a level (given as a row) in the standard basis
   for levels of the f^numE configuration.
The function admits the following options:
  \"Units\" : The units in which the line strengths are given. The
  default is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI
  \" then the unit of the line strength is (A m^2)^2 = (J/T)^2. If \
  "Hartree\" then the line strength is given in units of 2 \[Mu]B.";
Options[LevelMagDipoleLineStrength] = {
  "Units"->"SI"
  };
LevelMagDipoleLineStrength[theEigensys_List, numE0_Integer,
  OptionsPattern[]] := Module[
  {numE, levelMagOp, allEigenvecs, magDipoleLineStrength, units},
  (
    numE          = Min[14-numE0,numE0];
    levelMagOp    = LevelMagDipoleMatrixAssembly[numE];
    allEigenvecs  = Transpose[Last/@theEigensys];
    units         = OptionValue["Units"];
    magDipoleLineStrength         = Transpose[allEigenvecs].
  levelMagOp.allEigenvecs;
    magDipoleLineStrength         = Abs[magDipoleLineStrength]^2;
    Which[
      units=="SI",
        Return[4 \[Mu]B^2 * magDipoleLineStrength],
      units=="Hartree",
        Return[magDipoleLineStrength]
    ];
  )
];

LevelMagDipoleOscillatorStrength::usage = "
  LevelMagDipoleOscillatorStrength[eigenSys, numE] calculates the
  magnetic dipole oscillator strengths for an ion whose levels are
  described by eigenSys in configuration f^numE. The refractive
  index of the medium is relevant, but here it is assumed to be 1,
  this can be changed through the option \"RefractiveIndex\".
  eigenSys must consist of a lists of lists with three elements: the
   first element being the energy of the level, the second element
  being the J of the level, and the third element being the
  eigenvector of the level.
   The function returns a list with the following elements:
    - basis : A list with the allowed {SL, J} terms in the f^numE
  configuration. Equal to BasisLSJ[numE].
    - eigenSys : A list with the eigensystem of the Hamiltonian for
   the f^n configuration in the level description.
    - magDipoleOstrength : A square array whose elements represent
  the magnetic dipole oscillator strengths between the levels given
  in eigenSys such that the element magDipoleOstrength[[i,j]] is the
   oscillator strength between the levels |Subscript[\[Psi], i]> and
```

```
    |Subscript[\[Psi], j]>. In this array the elements below the
   diagonal represent emission oscillator strengths, and elements
   above the diagonal represent absorption oscillator strengths. The
   emission oscillator strengths are negative. The oscillator
   strength is a dimensionless quantity.
4180  The function admits the following option:
4181    \"RefractiveIndex\" : The refractive index of the medium where
   the transitions are taking place. This may be a number or a
   function. If a number then the oscillator strengths are calculated
    assuming a wavelength-independent refractive index as given. If a
    function then the refractive indices are calculated accordingly
   to the vaccum wavelength of each transition (the function must
   admit a single argument equal to the wavelength in nm). The
   default is 1.
4182  For reference see equation (27.8) in Rudzikas (2007). The
   expression for the line strenght is the simplest when using atomic
    units, (27.8) is missing a factor of α^2.";
4183 Options[LevelMagDipoleOscillatorStrength]={
4184   "RefractiveIndex" -> 1
4185 };
4186 LevelMagDipoleOscillatorStrength[eigenSys_, numE_, OptionsPattern
   []] := Module[
4187   {eigenEnergies, eigenVecs, levelJs,
4188   energyDiffs, magDipoleOstrength, nRef,
4189   wavelengthsInNM, nRefs, degenDivisor},
4190   (
4191     basis          = BasisLSJ[numE];
4192     eigenEnergies  = First/@eigenSys;
4193     nRef           = OptionValue["RefractiveIndex"];
4194     eigenVecs      = Last/@eigenSys;
4195     levelJs        = #[[2]]&/@eigenSys;
4196     energyDiffs    = -Outer[Subtract,eigenEnergies,eigenEnergies];
4197     energyDiffs  *= kayserToHartree;
4198     magDipoleOstrength = LevelMagDipoleLineStrength[eigenSys, numE,
    "Units"->"Hartree"];
4199     magDipoleOstrength = 2/3 * αFine^2 * energyDiffs *
   magDipoleOstrength;
4200     degenDivisor       = #1 / ( 2 * levelJs[[#2[[1]]]] + 1 ) &;
4201     magDipoleOstrength = MapIndexed[degenDivisor,
   magDipoleOstrength, {2}];
4202     Which[nRef===1,
4203       True,
4204       NumberQ[nRef],
4205         (
4206           magDipoleOstrength = nRef * magDipoleOstrength;
4207         ),
4208       True,
4209         (
4210           wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
    10^7;
4211           nRefs              = Map[nRef, wavelengthsInNM];
4212           magDipoleOstrength = nRefs * magDipoleOstrength;
4213         )
4214     ];
4215     Return[{basis, eigenSys, magDipoleOstrength}];
4216   )
4217 ];
4218
4219 LevelMagDipoleSpontaneousDecayRates::usage = "
   LevelMagDipoleSpontaneousDecayRates[eigenSys, numE] calculates the
    spontaneous emission rates for the magnetic dipole transitions
   between the levels given in eigenSys. The function returns a
   square array whose elements represent the spontaneous emission
   rates between the levels given in eigenSys such that the element
   [[i,j]] of the returned array is the rate of spontaneous emission
   from the level |Subscript[\[Psi], i]> to the level |Subscript[\[
   Psi], j]>. In this array the elements below the diagonal represent
    emission rates, and elements above the diagonal are identically
   zero.
4220 The function admits two optional arguments:
4221   + \"Units\" : The units in which the rates are given. The default
    is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI\" then
    the rates are given in s^-1. If \"Hartree\" then the rates are
   given in the atomic unit of frequency.
4222   + \"RefractiveIndex\" : The refractive index of the medium where
   the transitions are taking place. This may be a number or a
```

```
    function. If a number then the rates are calculated assuming a
    wavelength-independent refractive index as given. If a function
    then the refractive indices are calculated accordingly to the
    vaccum wavelength of each transition (the function must admit a
    single argument equal to the wavelength in nm). The default is 1."
    ;
4223  Options[LevelMagDipoleSpontaneousDecayRates] = {
4224    "Units" -> "SI",
4225    "RefractiveIndex" -> 1};
4226  LevelMagDipoleSpontaneousDecayRates[eigenSys_List, numE_Integer,
      OptionsPattern[]] := Module[
4227    {levMDlineStrength, eigenEnergies, energyDiffs,
4228    levelJs, spontaneousRatesInHartree, spontaneousRatesInSI,
4229    degenDivisor, units, nRef, nRefs, wavelengthsInNM},
4230    (
4231      nRef              = OptionValue["RefractiveIndex"];
4232      units             = OptionValue["Units"];
4233      levMDlineStrength =
      LowerTriangularize@LevelMagDipoleLineStrength[eigenSys,numE,"Units
      "->"Hartree"];
4234      levMDlineStrength = SparseArray[levMDlineStrength];
4235      eigenEnergies     = First /@ eigenSys;
4236      energyDiffs       = Outer[Subtract, eigenEnergies,
      eigenEnergies];
4237      energyDiffs       = kayserToHartree * energyDiffs;
4238      energyDiffs       = SparseArray[LowerTriangularize[energyDiffs
      ]];
4239      levelJs           = #[[2]]& /@ eigenSys;
4240      spontaneousRatesInHartree = 4/3 αFine^5 * energyDiffs^3 *
      levMDlineStrength;
4241      degenDivisor              = #1 / (2*levelJs[[#2[[1]]]] + 1)&;
4242      spontaneousRatesInHartree = MapIndexed[degenDivisor,
      spontaneousRatesInHartree, {2}];
4243      Which[nRef===1,
4244        True,
4245        NumberQ[nRef],
4246          (
4247            spontaneousRatesInHartree = nRef^3 *
      spontaneousRatesInHartree;
4248          ),
4249        True,
4250          (
4251            wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
       10^7;
4252            nRefs              = Map[nRef, wavelengthsInNM];
4253            spontaneousRatesInHartree = nRefs^3 *
      spontaneousRatesInHartree;
4254          )
4255      ];
4256      If[units == "SI",
4257        (
4258          spontaneousRatesInSI = 1/hartreeTime *
      spontaneousRatesInHartree;
4259          Return[SparseArray@spontaneousRatesInSI];
4260        ),
4261        Return[SparseArray@spontaneousRatesInHartree];
4262      ];
4263    )
4264  ];
4265
4266  (* ############## Optical Transitions for Levels ############## *)
4267  (* ############################################################ *)
4268
4269  (* ############################################################ *)
4270  (* ################### Eigensystem analysis ################### *)
4271
4272  PrettySaundersSL::usage = "PrettySaundersSL[SL] produces a human-
      readeable symbol for the spectroscopic term SL. SL can be either a
       string (in RS notation for the term) or a list of two numbers {S,
       L}. The option \"Representation\" can be used to specify whether
      the output is given as a symbol or as a ket. The default is \"Ket\
      ".";
4273  Options[PrettySaundersSL] = {"Representation" -> "Ket"};
4274  PrettySaundersSL[SL_, OptionsPattern[]] := (
4275    If[StringQ[SL],
4276    (
```

```
4277        {S,L} = FindSL[SL];
4278        L     = StringTake[SL,{2,-1}];
4279      ),
4280        {S,L}=SL
4281      ];
4282      pretty = RowBox[{
4283          AdjustmentBox[Style[2*S+1,Smaller], BoxBaselineShift->-1,
          BoxMargins->0],
4284          AdjustmentBox[PrintL[L]]
4285        }
4286      ];
4287      pretty = DisplayForm[pretty];
4288      pretty = Which[
4289        OptionValue["Representation"]=="Ket",
4290          Ket[{pretty}],
4291        OptionValue["Representation"]=="Symbol",
4292          pretty
4293      ];
4294      Return[pretty];
4295    );
4296
4297    PrettySaundersSLJmJ::usage = "PrettySaundersSLJmJ[{SL, J, mJ}]
        produces a formatted symbol for the given basis vector {SL, J, mJ
        }.";
4298    Options[PrettySaundersSLJmJ] = {"Representation" -> "Ket"};
4299    PrettySaundersSLJmJ[{SL_, J_, mJ_}, OptionsPattern[]] := (If[
4300      StringQ[SL],
4301      ({S, L} = FindSL[SL];
4302        L = StringTake[SL, {2, -1}];
4303        ),
4304      {S, L} = SL];
4305      pretty = RowBox[{AdjustmentBox[Style[2*S + 1, Smaller],
4306          BoxBaselineShift -> -1, BoxMargins -> 0],
4307          AdjustmentBox[PrintL[L], BoxMargins -> -0.2],
4308          AdjustmentBox[
4309          Style[Row[{InputForm[J], ",", mJ}], Small],
4310          BoxBaselineShift -> 1,
4311          BoxMargins -> {{0.7, 0}, {0.4, 0.4}}]}];
4312      pretty = DisplayForm[pretty];
4313      If[OptionValue["Representation"] == "Ket",
4314        pretty = Ket[{pretty}]
4315      ];
4316      Return[pretty];
4317    );
4318
4319    PrettySaundersSLJ::usage = "PrettySaundersSLJ[{SL, J}] produces a
        formatted symbol for the given level {SL, J}. SL can be either a
        list of two numbers representing S and L or a string representing
        the spin multiplicity in spectroscopic notation. J is the total
        angular momentum to which S and L have coupled to. The option \"
        Representation\" can be used to specify whether the output is
        given as a symbol or as a ket. The default is \"Ket\".";
4320    Options[PrettySaundersSLJ] = {"Representation"->"Ket"};
4321    PrettySaundersSLJ[{SL_,J_},OptionsPattern[]] := (
4322      If[StringQ[SL],
4323      (
4324        {S,L}=FindSL[SL];
4325        L=StringTake[SL,{2,-1}];
4326      ),
4327        {S,L}=SL
4328      ];
4329      pretty = RowBox[{
4330          AdjustmentBox[Style[2*S+1,Smaller],BoxBaselineShift->-1,
          BoxMargins->0],
4331          AdjustmentBox[PrintL[L],BoxMargins->-0.2],
4332          AdjustmentBox[Style[InputForm[J],Small,FontTracking->"Narrow"
          ],BoxBaselineShift->1,BoxMargins->{{0.7,0},{0.4,0.4}}]
4333        }
4334      ];
4335      pretty = DisplayForm[pretty];
4336      pretty = Which[
4337        OptionValue["Representation"]=="Ket",
4338          Ket[{pretty}],
4339        OptionValue["Representation"]=="Symbol",
4340          pretty
4341      ];
```

```
    Return[pretty];
  );

  BasisVecInRusselSaunders::usage = "BasisVecInRusselSaunders[
    basisVec] takes a basis vector in the format {LSstring, Jval,
    mJval} and returns a formatted symbol for the corresponding Russel
    -Saunders term.";
  BasisVecInRusselSaunders[basisVec_] := (
    {LSstring, Jval, mJval} = basisVec;
    Ket[PrettySaundersSLJmJ[basisVec]]
  );

  LSJMJTemplate =
    StringTemplate[
    "\!\(\*TemplateBox[{\nRowBox[{\"`LS`\", \",\", \nRowBox[{\"J\", \
  \"=\", \"`J`\"}], \",\", \nRowBox[{\"mJ\", \"=\", \"`mJ`\"}]}]},\n\
  \"Ket\"]\)"];

  BasisVecInLSJMJ::usage = "BasisVecInLSJMJ[basisVec] takes a basis
    vector in the format {{{LSstring, Jval}, mJval}, nucSpin} and
    returns a formatted symbol for the corresponding LSJMJ term in the
     form |LS, J=..., mJ=...>.";
  BasisVecInLSJMJ[basisVec_] := (
    {LSstring, Jval, mJval} = basisVec;
    LSJMJTemplate[<|
      "LS" -> LSstring,
      "J" -> ToString[Jval, InputForm],
      "mJ" -> ToString[mJval, InputForm]|>]
  );

  ParseStates::usage = "ParseStates[eigenSys, basis] takes a list of
    eigenstates in terms of their coefficients in the given basis and
    returns a list of the same states in terms of their energy, LSJMJ
    symbol, J, mJ, S, L, LSJ symbol, and LS symbol. eigenSys is a list
     of lists with two elements, in each list the first element is the
     energy and the second one the corresponding eigenvector. The LS
    symbol returned corresponds to the term with the largest
    coefficient in the given basis.";
  ParseStates[states_, basis_, OptionsPattern[]] := Module[
    {parsedStates},
    (
      parsedStates = Table[(
        {energy, eigenVec}      = state;
        maxTermIndex            = Ordering[Abs[eigenVec]][[-1]];
        {LSstring, Jval, mJval} = basis[[maxTermIndex]];
        LSJsymbol               = Subscript[LSstring, {Jval, mJval}];
        LSJMJsymbol             = LSstring <> ToString[Jval,
    InputForm];
        {S, L}                  = FindSL[LSstring];
        {energy, LSstring, Jval, mJval, S, L, LSJsymbol, LSJMJsymbol}
      ),
      {state, states}
      ];
      Return[parsedStates];
    )
  ];

  ParseStatesByNumBasisVecs::usage = "ParseStatesByNumBasisVecs[
    eigenSys, basis, numBasisVecs, roundTo] takes a list of
    eigenstates (given in eigenSys) in terms of their coefficients in
    the given basis and returns a list of the same states in terms of
    their energy and the coefficients at most numBasisVecs basis
    vectors. By default roundTo is 0.01 and this is the value used to
    round the amplitude coefficients. eigenSys is a list of lists with
     two elements, in each list the first element is the energy and
    the second one the corresponding eigenvector.
  The option \"Coefficients\" can be used to specify whether the
    coefficients are given as \"Amplitudes\" or \"Probabilities\". The
     default is \"Amplitudes\".
  ";
  Options[ParseStatesByNumBasisVecs] = {
    "Coefficients" -> "Amplitudes",
    "Representation" -> "Ket",
    "ReturnAs" -> "Dot"
    };
  ParseStatesByNumBasisVecs[eigensys_List, basis_List,
```

```
  numBasisVecs_Integer, roundTo_Real : 0.01, OptionsPattern[]] :=
  Module[
4394  {parsedStates, energy, eigenVec,
4395  probs, amplitudes, ordering,
4396  returnAs,
4397  chosenIndices, majorComponents,
4398  majorAmplitudes, majorRep},
4399  (
4400    returnAs     = OptionValue["ReturnAs"];
4401    parsedStates = Table[(
4402      {energy, eigenVec} = state;
4403      energy             = Chop[energy];
4404      probs              = Round[Abs[eigenVec^2], roundTo];
4405      amplitudes         = Round[eigenVec, roundTo];
4406      ordering           = Ordering[probs];
4407      chosenIndices      = ordering[[-numBasisVecs ;;]];
4408      majorComponents    = basis[[chosenIndices]];
4409      majorThings = If[OptionValue["Coefficients"] == "
  Probabilities",
4410        (
4411          probs[[chosenIndices]]
4412        ),
4413        (
4414          amplitudes[[chosenIndices]]
4415        )
4416      ];
4417      majorComponents    = PrettySaundersSLJmJ[#, "Representation"
  -> OptionValue["Representation"]] & /@ majorComponents;
4418      nonZ               = (# != 0.) & /@ majorThings;
4419      majorThings        = Pick[majorThings, nonZ];
4420      majorComponents    = Pick[majorComponents, nonZ];
4421      If[OptionValue["Coefficients"] == "Probabilities",
4422        (
4423          majorThings = majorThings * 100 * "%";
4424        )
4425      ];
4426      majorRep  = Which[
4427                  returnAs == "Dot",
4428                    majorThings . majorComponents,
4429                  returnAs == "List",
4430                    Transpose[{Reverse@majorThings,
  Reverse@majorComponents}]
4431                  ];
4432      {energy, majorRep}
4433      ),
4434      {state, eigensys}];
4435    Return[parsedStates]
4436  )
4437 ];
4438
4439 FindThresholdPosition::usage = "FindThresholdPosition[list,
    threshold] returns the position of the first element in list that
    is greater than or equal to threshold. If no such element exists,
    it returns the length of list. The elements of the given list must
     be in ascending order.";
4440 FindThresholdPosition[list_, threshold_] := Module[
4441   {position},
4442   (
4443   position = Position[list, _?(# >= threshold &), 1, 1];
4444   thrPos = If[Length[position] > 0,
4445     position[[1, 1]],
4446     Length[list]];
4447   If[thrPos == 0,
4448     Return[1],
4449     Return[thrPos]]
4450   )
4451 ];
4452
4453 ParseStatesByProbabilitySum::usage = "ParseStatesByProbabilitySum[
    eigensys, basis, probSum] takes a list of eigenstates in terms of
    their coefficients in the given basis and returns a list of the
    same states in terms of their energy and the coefficients of the
    basis vectors that sum to at least probSum.";
4454 ParseStatesByProbabilitySum[eigensys_, basis_, probSum_, roundTo_ :
     0.01, maxParts_: 20] := Module[
4455   {parsedByProb, numStates, state, energy,
```

```
4456      eigenVec, amplitudes, probs, ordering,
4457      orderedProbs, truncationIndex, accProb,
4458      thresholdIndex, chosenIndices, majorComponents,
4459      majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep
          },
4460      (
4461        numStates    = Length[eigensys];
4462        parsedByProb = Table[(
4463          state              = eigensys[[idx]];
4464          {energy, eigenVec} = state;
4465          (*Round them up*)
4466          amplitudes         = Round[eigenVec, roundTo];
4467          probs              = Round[Abs[eigenVec^2], roundTo];
4468          ordering           = Reverse[Ordering[probs]];
4469          (*Order the probabilities from high to low*)
4470          orderedProbs       = probs[[ordering]];
4471          (*To speed up Accumulate, assume that only as much as
            maxParts will be needed*)
4472          truncationIndex    = Min[maxParts, Length[orderedProbs]];
4473          orderedProbs       = orderedProbs[[;;truncationIndex]];
4474          (*Accumulate the probabilities*)
4475          accProb            = Accumulate[orderedProbs];
4476          (*Find the index of the first element in accProb that is
            greater than probSum*)
4477          thresholdIndex     = Min[Length[accProb],
            FindThresholdPosition[accProb, probSum]];
4478          (*Grab all the indicees up till that one*)
4479          chosenIndices      = ordering[[;; thresholdIndex]];
4480          (*Select the corresponding elements from the basis*)
4481          majorComponents    = basis[[chosenIndices]];
4482          (*Select the corresponding amplitudes*)
4483          majorAmplitudes    = amplitudes[[chosenIndices]];
4484          (*Take their absolute value*)
4485          absMajorAmplitudes = Abs[majorAmplitudes];
4486          (*Make sure that there are no effectively zero contributions
            *)
4487          notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_
            /; x != 0]];
4488          (* majorComponents    = PrettySaundersSLJmJ
            [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
4489          majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
4490          majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
4491          majorComponents    = majorComponents[[notnullAmplitudes]];
4492          (*Multiply and add to build the final Ket*)
4493          majorRep           = majorAmplitudes . majorComponents;
4494          {energy, majorRep}
4495          ), {idx, numStates}];
4496        Return[parsedByProb];
4497      )
4498    ];
4499
4500    (* ################## Eigensystem analysis ################### *)
4501    (* ############################################################ *)
4502
4503    (* ############################################################ *)
4504    (* ########################### Misc ########################### *)
4505
4506    SymbToNum::usage = "SymbToNum[expr, numAssociation] takes an
          expression expr and returns what results after making the
          replacements defined in the given replacementAssociation. If
          replacementAssociation doesn't define values for expected keys,
          they are taken to be zero.";
4507    SymbToNum[expr_, replacementAssociation_] := (
4508      includedKeys = Keys[replacementAssociation];
4509      (*If a key is not defined, make its value zero.*)
4510      fullAssociation = Table[(
4511        If[MemberQ[includedKeys, key],
4512          ToExpression[key]->replacementAssociation[key],
4513          ToExpression[key]->0
4514        ]
4515      ),
4516      {key, paramSymbols}];
4517      Return[expr/.fullAssociation];
4518    );
4519
4520    SimpleConjugate::usage = "SimpleConjugate[expr] takes an expression
```

```
    and applies a simplified version of the conjugate in that all it
    does is that it replaces the imaginary unit I with -I. It assumes
    that every other symbol is real so that it remains the same under
    complex conjugation. Among other expressions it is valid for any
    rational or polynomial expression with complex coefficients and
    real variables.";
4521  SimpleConjugate[expr_] := expr /. Complex[a_, b_] :> a - I b;
4522
4523  ExportMZip::usage = "ExportMZip[\"dest.[zip,m]\", expr] saves a
    compressed version of expr to the given destination.";
4524  ExportMZip[filename_, expr_] := Module[
4525    {baseName, exportName, mImportName, zipImportName},
4526    (
4527      baseName    = FileBaseName[filename];
4528      exportName  = StringReplace[filename,".m"->".zip"];
4529      mImportName = StringReplace[exportName,".zip"->".m"];
4530      If[FileExistsQ[mImportName],
4531      (
4532        PrintTemporary[mImportName<>" exists already, deleting"];
4533        DeleteFile[mImportName];
4534        Pause[2];
4535      )
4536      ];
4537      Export[exportName, (baseName<>".m") -> expr];
4538    )
4539  ];
4540
4541  ImportMZip::usage = "ImportMZip[filename] imports a .m file inside
    a .zip file with corresponding filename. If the Option \"Leave
    Uncompressed\" is set to True (the default) then this function
    also leaves an umcompressed version of the object in the same
    folder of filename";
4542  Options[ImportMZip]={"Leave Uncompressed" -> True};
4543  ImportMZip[filename_String, OptionsPattern[]] := Module[
4544    {baseName, importKey, zipImportName, mImportName, mxImportName,
    imported},
4545    (
4546      baseName      = FileBaseName[filename];
4547      (*Function allows for the filename to be .m or .zip*)
4548      importKey     = baseName <> ".m";
4549      zipImportName = StringReplace[filename, ".m"->".zip"];
4550      mImportName   = StringReplace[zipImportName, ".zip"->".m"];
4551      mxImportName  = StringReplace[zipImportName, ".zip"->".mx"];
4552      Which[
4553        FileExistsQ[mxImportName],
4554      (
4555        PrintTemporary[".mx version exists already, importing that
    instead ..."];
4556        Return[Import[mxImportName]];
4557      ),
4558        FileExistsQ[mImportName],
4559      (
4560        PrintTemporary[".m version exists already, importing that
    instead ..."];
4561        imported = Import[mImportName];
4562        If[OptionValue["Leave Uncompressed"],
4563          (
4564            Export[mxImportName, imported];
4565          )
4566        ];
4567        Return[Import[mImportName]];
4568      )
4569      ];
4570      imported = Import[zipImportName, importKey];
4571      If[OptionValue["Leave Uncompressed"],
4572        (
4573          Export[mImportName, imported];
4574          Export[mxImportName, imported];
4575        )
4576      ];
4577      Return[imported];
4578    )
4579  ];
4580
4581  ReplaceInSparseArray::usage = "ReplaceInSparseArray[sparseArray,
    rules] takes a sparse array that may contain symbolic quantities
```

```
    and returns a sparse array in which the given rules have been used
     on every element.";
4582  ReplaceInSparseArray[sparseA_SparseArray, rules_] := (
4583    SparseArray[Automatic,
4584      Dimensions[sparseA],
4585      sparseA["Background"] /. rules,
4586      {
4587        1,
4588        {sparseA["RowPointers"], sparseA["ColumnIndices"]},
4589        sparseA["NonzeroValues"] /. rules
4590      }
4591      ]
4592  );
4593
4594  MapToSparseArray::usage = "MapToSparseArray[sparseArray, function]
       takes a sparse array and returns a sparse array after the function
     has been applied to it.";
4595  MapToSparseArray[sparseA_SparseArray, func_] := Module[
4596    {nonZ, backg, mapped},
4597    (
4598      nonZ   = func /@ sparseA["NonzeroValues"];
4599      backg  = func[sparseA["Background"]];
4600      mapped = SparseArray[Automatic,
4601        Dimensions[sparseA],
4602        backg,
4603        {
4604          1,
4605          {sparseA["RowPointers"], sparseA["ColumnIndices"]},
4606          nonZ
4607        }
4608      ];
4609      Return[mapped];
4610    )
4611  ];
4612
4613  ParseTeXLikeSymbol::usage = "ParseTeXLikeSymbol[string] parses a
       string for a symbol given in LaTeX notation and returns a
       corresponding mathematica symbol. The string may have expressions
       for several symbols, they need to be separated by single spaces.
       In addition the _ and ^ symbols used in LaTeX notation need to
       have arguments that are enclosed in parenthesis, for example \"x_2
       \" is invalid, instead \"x_{2}\" should have been given.";
4614  Options[ParseTeXLikeSymbol] = {"Form" -> "List"};
4615  ParseTeXLikeSymbol[bigString_, OptionsPattern[]] := Module[
4616    {form, mainSymbol, symbols},
4617    (
4618      form = OptionValue["Form"];
4619      (* parse greek *)
4620      symbols = Table[(
4621          str = StringReplace[string, {"\\alpha" -> "α",
4622            "\\beta" -> "β",
4623            "\\gamma" -> "γ",
4624            "\\psi" -> "\[Psi]"}];
4625          symbol = Which[
4626            StringContainsQ[str, "_"] && Not[StringContainsQ[str, "^"
       ]],
4627            (
4628              (*yes sub no sup*)
4629              mainSymbol = StringSplit[str, "_"][[1]];
4630              mainSymbol = ToExpression[mainSymbol];
4631
4632              subPart =
4633                StringCases[str,
4634                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
4635              Subscript[mainSymbol, subPart]
4636            ),
4637            Not[StringContainsQ[str, "_"]] && StringContainsQ[str, "^
       "],
4638            (
4639              (*no sub yes sup*)
4640              mainSymbol = StringSplit[str, "^"][[1]];
4641              mainSymbol = ToExpression[mainSymbol];
4642
4643              supPart =
4644                StringCases[str,
4645                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
```

```
              Superscript[mainSymbol, supPart]
            ),
            StringContainsQ[str, "_"] && StringContainsQ[str, "^"],
            (
              (*yes sub yes sup*)
              mainSymbol = StringSplit[str, "_"][[1]];
              mainSymbol = ToExpression[mainSymbol];
              {subPart, supPart} =
                StringCases[str, RegularExpression@"\\{(.*?)\\}" -> "$1"];
              Subsuperscript[mainSymbol, subPart, supPart]
            ),
            True,
            (
              (*no sup or sub*)
              str
            )
            ];
          symbol
          ),
        {string, StringSplit[bigString, " "]}
      ];
      Which[
        form == "Row",
        Return[Row[symbols]],
        form == "List",
        Return[symbols]
      ]
    )
  ];

  FromArrayToTable::usage = "FromArrayToTable[array, labels, energies] takes a square array of values and returns a table with the labels of the rows and columns, the energies of the initial and final levels, the level energies, the vacuum wavelength of the transition, and the value of the array. The array must be square and the labels and energies must be compatible with the order implied by the array. The array must be a square array of values. The function returns a list of lists with the following elements:
    - Initial level index
    - Final level index
    - Initial level label
    - Final level label
    - Initial level energy
    - Final level energy
    - Vacuum wavelength
    - Value of the array element.
    - The reciprocal of the value of the array element.
  Elements in which the array is zero are not included in the return of this function.";
  FromArrayToTable[array_,labels_,energies_] := Module[
    {tableFun, atl},
    (
      tableFun = {
        #2[[1]],
        #2[[2]],
        labels[[#2[[1]]]],
        labels[[#2[[2]]]],
        energies[[#2[[1]]]],
        energies[[#2[[2]]]],
        If[#2[[1]]==#2[[2]],"--",10^7/(energies[[#2[[1]]]]-energies[[#2[[2]]]])],
        #1
      }&;
      atl = Select[Flatten[MapIndexed[tableFun,array,{2}],1],#[[-1]]=!=0.&];
      atl = Append[#,1/#[[-1]]]&/@atl;
      Return[atl]
    )
  ];
  (* ########################## Misc ########################## *)
  (* ########################################################## *)

  (* ########################################################## *)
  (* ################## Some Plotting Routines ################## *)
```

```
EnergyLevelDiagram::usage = "EnergyLevelDiagram[states] takes states and produces a visualization of its energy spectrum.
The resultant visualization can be navigated by clicking and dragging to zoom in on a region, or by clicking and dragging horizontally while pressing Ctrl. Double-click to reset the view.";
Options[EnergyLevelDiagram] = {
  "Title"->"",
  "ImageSize"->1000,
  "AspectRatio" -> 1/8,
  "Background"->"Automatic",
  "Epilog"->{},
  "Explorer"->True,
  "Energy Unit" -> "cm^-1"
  };
EnergyLevelDiagram[states_, OptionsPattern[]] := Module[
  {energies, epi, explora},
  (
    energies = If[Length[Dimensions[states]]==1,
      states,
      First/@states
    ];
    epi = OptionValue["Epilog"];
    explora = If[OptionValue["Explorer"],
      ExploreGraphics,
      Identity
    ];
    frameLabel = "E (" <> OptionValue["Energy Unit"] <> ")";
    plotTips = Which[
      OptionValue["Energy Unit"] == "cm^-1",
      Tooltip[{{#, 0}, {#, 1}}, {Quantity[#/8065.54429, "eV"], Quantity[#, 1/"Centimeters"]}] &/@ energies,
      OptionValue["Energy Unit"] == "eV",
      Tooltip[{{#, 0}, {#, 1}}, {Quantity[# * 8065.54429, 1/"Centimeters"], Quantity[#, "eV"]}] &/@ energies,
      True,
      Tooltip[{{#, 0}, {#, 1}}, Quantity[# , OptionValue["Energy Unit"]]] &/@ energies
    ];
    explora@ListPlot[plotTips,
      Joined       -> True,
      PlotStyle    -> Black,
      AspectRatio  -> OptionValue["AspectRatio"],
      ImageSize    -> OptionValue["ImageSize"],
      Frame        -> True,
      PlotRange    -> {All, {0, 1}},
      FrameTicks   -> {{None, None}, {Automatic, Automatic}},
      FrameStyle   -> Directive[15, Dashed, Thin],
      PlotLabel    -> Style[OptionValue["Title"], 15, Bold],
      Background   -> OptionValue["Background"],
      FrameLabel   -> {frameLabel},
      Epilog       -> epi]
  )
];

ExploreGraphics::usage = "Pass a Graphics object to explore it. Zoom by clicking and dragging a rectangle. Pan by clicking and dragging while pressing Ctrl. Click twice to reset view.
Based on ZeitPolizei @ https://mathematica.stackexchange.com/questions/7142/how-to-manipulate-2d-plots.
The option \"OptAxesRedraw\" can be used to specify whether the axes should be redrawn. The default is False.";
Options[ExploreGraphics] = {OptAxesRedraw -> False};
ExploreGraphics[graph_Graphics, opts : OptionsPattern[]] := With[
  {
    gr  = First[graph],
    opt = DeleteCases[Options[graph],
          PlotRange -> PlotRange | AspectRatio | AxesOrigin -> _],
    plr = PlotRange /. AbsoluteOptions[graph, PlotRange],
    ar  = AspectRatio /. AbsoluteOptions[graph, AspectRatio],
    ao  = AbsoluteOptions[AxesOrigin],
    rectangle = {Dashing[Small],
      Line[{#1,
            {First[#2], Last[#1]},
            #2,
            {First[#1], Last[#2]},
```

```
4776                #1}]} &,
4777      optAxesRedraw = OptionValue[OptAxesRedraw]
4778    },
4779    DynamicModule[
4780        {dragging=False, first, second, rx1, rx2, ry1, ry2,
4781        range = plr},
4782        {{rx1, rx2}, {ry1, ry2}} = plr;
4783      Panel@
4784      EventHandler[
4785        Dynamic@Graphics[
4786          If[dragging, {gr, rectangle[first, second]}, gr],
4787          PlotRange -> Dynamic@range,
4788          AspectRatio -> ar,
4789          AxesOrigin -> If[optAxesRedraw,
4790            Dynamic@Mean[range\[Transpose]], ao],
4791          Sequence @@ opt],
4792        {{"MouseDown", 1} :> (
4793          first = MousePosition["Graphics"]
4794          ),
4795        {"MouseDragged", 1} :> (
4796          dragging = True;
4797          second = MousePosition["Graphics"]
4798          ),
4799        "MouseClicked" :> (
4800          If[CurrentValue@"MouseClickCount"==2,
4801            range = plr];
4802          ),
4803        {"MouseUp", 1} :> If[dragging,
4804          dragging = False;
4805
4806          range = {{rx1, rx2}, {ry1, ry2}} =
4807            Transpose@{first, second};
4808          range[[2]] = {0, 1}],
4809        {"MouseDown", 2} :> (
4810          first = {sx1, sy1} = MousePosition["Graphics"]
4811          ),
4812        {"MouseDragged", 2} :> (
4813          second = {sx2, sy2} = MousePosition["Graphics"];
4814          rx1 = rx1 - (sx2 - sx1);
4815          rx2 = rx2 - (sx2 - sx1);
4816          ry1 = ry1 - (sy2 - sy1);
4817          ry2 = ry2 - (sy2 - sy1);
4818          range = {{rx1, rx2}, {ry1, ry2}};
4819          range[[2]] = {0, 1};
4820          )}]]];
4821
4822  LabeledGrid::usage = "LabeledGrid[data, rowHeaders, columnHeaders]
       provides a grid of given data interpreted as a matrix of values
       whose rows are labeled by rowHeaders and whose columns are labeled
        by columnHeaders. When hovering with the mouse over the grid
       elements, the row and column labels are displayed with the given
       separator between them.";
4823  Options[LabeledGrid]={
4824      ItemSize->Automatic,
4825      Alignment->Center,
4826      Frame->All,
4827      "Separator"->",",
4828      "Pivot"->""
4829  };
4830  LabeledGrid[data_,rowHeaders_,columnHeaders_,OptionsPattern[]] :=
       Module[
4831    {gridList=data, rowHeads=rowHeaders, colHeads=columnHeaders},
4832    (
4833      separator=OptionValue["Separator"];
4834      pivot=OptionValue["Pivot"];
4835      gridList=Table[
4836              Tooltip[
4837                data[[rowIdx,colIdx]],
4838                DisplayForm[
4839                  RowBox[{rowHeads[[rowIdx]],
4840                          separator,
4841                          colHeads[[colIdx]]}
4842                        ]
4843                        ]
4844                      ],
4845          {rowIdx,Dimensions[data][[1]]},
```

```
4846            {colIdx,Dimensions[data][[2]]}];
4847        gridList=Transpose[Prepend[gridList,colHeads]];
4848        rowHeads=Prepend[rowHeads,pivot];
4849        gridList=Prepend[gridList,rowHeads]//Transpose;
4850        Grid[gridList,
4851            Frame->OptionValue[Frame],
4852            Alignment->OptionValue[Alignment],
4853            Frame->OptionValue[Frame],
4854            ItemSize->OptionValue[ItemSize]
4855            ]
4856     )
4857   ];
4858
4859   HamiltonianForm::usage = "HamiltonianForm[hamMatrix, basisLabels]
       takes the matrix representation of a hamiltonian together with a
       set of symbols representing the ordered basis in which the
       operator is represented. With this it creates a displayed form
       that has adequately labeled row and columns together with
       informative values when hovering over the matrix elements using
       the mouse cursor.";
4860   Options[HamiltonianForm]={"Separator"->"","Pivot"->""};
4861   HamiltonianForm[hamMatrix_, basisLabels_List, OptionsPattern[]] :=
       (
4862      braLabels=DisplayForm[RowBox[{"\[LeftAngleBracket]",#,"\[
       RightBracketingBar]"}]]& /@ basisLabels;
4863      ketLabels=DisplayForm[RowBox[{"\[LeftBracketingBar]",#,"\[
       RightAngleBracket]"}]]& /@ basisLabels;
4864      LabeledGrid[hamMatrix,braLabels,ketLabels,"Separator"->
       OptionValue["Separator"],"Pivot"->OptionValue["Pivot"]]
4865      )
4866
4867   HamiltonianMatrixPlot::usage = "HamiltonianMatrixPlot[hamMatrix,
       basisLabels] creates a matrix plot of the given hamiltonian matrix
        with the given basis labels. The matrix elements can be hovered
       over to display the corresponding row and column labels together
       with the value of the matrix element. The option \"OverlayValues\"
        can be used to specify whether the matrix elements should be
       displayed on top of the matrix plot.";
4868   Options[HamiltonianMatrixPlot] = Join[Options[MatrixPlot], {"Hover"
       -> True, "OverlayValues" -> True}];
4869   HamiltonianMatrixPlot[hamMatrix_, basisLabels_, opts :
       OptionsPattern[]] := (
4870    braLabels = DisplayForm[RowBox[{"\[LeftAngleBracket]", #, "\[
       RightBracketingBar]"}]] & /@ basisLabels;
4871    ketLabels = DisplayForm[Rotate[RowBox[{"\[LeftBracketingBar]", #,
        "\[RightAngleBracket]"}],\[Pi]/2]] & /@ basisLabels;
4872    ketLabelsUpright = DisplayForm[RowBox[{"\[LeftBracketingBar]", #,
        "\[RightAngleBracket]"}]] & /@ basisLabels;
4873    numRows = Length[hamMatrix];
4874    numCols = Length[hamMatrix[[1]]];
4875    epiThings = Which[
4876      And[OptionValue["Hover"], Not[OptionValue["OverlayValues"]]],
4877      Flatten[
4878        Table[
4879          Tooltip[
4880            {
4881              Transparent,
4882              Rectangle[
4883              {j - 1, numRows - i},
4884              {j - 1, numRows - i} + {1, 1}
4885              ]
4886            },
4887          Row[{braLabels[[i]],ketLabelsUpright[[j]],"=",hamMatrix[[i,
       j]]}]
4888          ],
4889        {i, 1, numRows},
4890        {j, 1, numCols}
4891        ]
4892      ],
4893      And[OptionValue["Hover"], OptionValue["OverlayValues"]],
4894      Flatten[
4895        Table[
4896          Tooltip[
4897            {
4898              Transparent,
4899              Rectangle[
```

```
              {j - 1, numRows - i},
              {j - 1, numRows - i} + {1, 1}
              ]
            },
          DisplayForm[RowBox[{"\[LeftAngleBracket]", basisLabels[[i
  ]], "\[LeftBracketingBar]", basisLabels[[j]], "\[RightAngleBracket
  ]"}]]
          ],
        {i, numRows},
        {j, numCols}
        ]
      ],
      True,
      {}
      ];
    textOverlay = If[OptionValue["OverlayValues"],
      (
        Flatten[
        Table[
          Text[hamMatrix[[i, j]],
            {j - 1/2, numRows - i + 1/2}
          ],
        {i, 1, numRows},
        {j, 1, numCols}
        ]
        ]
      ),
      {}
      ];
    epiThings = Join[epiThings, textOverlay];
    MatrixPlot[hamMatrix,
      FrameTicks -> {
        {Transpose[{Range[Length[braLabels]], braLabels}], None},
        {None, Transpose[{Range[Length[ketLabels]], ketLabels}]}
        },
      Evaluate[FilterRules[{opts}, Options[MatrixPlot]]],
      Epilog -> epiThings
    ]
  );

  (* ################## Some Plotting Routines ################## *)
  (* ############################################################ *)

  (* ############################################################ *)
  (* ###################### Load Functions ###################### *)

  LoadAll::usage = "LoadAll[] executes most Load* functions.";
  LoadAll[] := (
    LoadTermLabels[];
    LoadCFP[];
    LoadUk[];
    LoadV1k[];
    LoadT22[];
    LoadSOOandECSOLS[];

    LoadElectrostatic[];
    LoadSpinOrbit[];
    LoadSOOandECSO[];
    LoadSpinSpin[];
    LoadThreeBody[];
    LoadChenDeltas[];
    LoadCarnall[];
  );

  fnTermLabels::usage = "This list contains the labels of f^n
    configurations. Each element of the list has four elements {LS,
    seniority, W, U}. At first sight this seems to only include the
    labels for the f^6 and f^7 configuration, however, all is included
     in these two.";

  LoadTermLabels::usage = "LoadTermLabels[] loads into the session
    the labels for the terms in the f^n configurations.";
  LoadTermLabels[] := (
    If[ValueQ[fnTermLabels], Return[]];
    PrintTemporary["Loading data for state labels in the f^n
    configurations..."];
```

```
4968    fnTermsFname = FileNameJoin[{moduleDir, "data", "fnTerms.m"}];
4969
4970    If[!FileExistsQ[fnTermsFname],
4971      (PrintTemporary[">> fnTerms.m not found, generating ..."];
4972        fnTermLabels = ParseTermLabels["Export"->True];
4973      ),
4974      fnTermLabels = Import[fnTermsFname];
4975    ];
4976  );
4977
4978  Carnall::usage = "Association of data from Carnall et al (1989)
       with the following keys: {data, annotations, paramSymbols,
       elementNames, rawData, rawAnnotations, annnotatedData, appendix:Pr
       :Association, appendix:Pr:Calculated, appendix:Pr:RawTable,
       appendix:Headings}";
4979
4980  LoadCarnall::usage = "LoadCarnall[] loads data for trivalent
       lanthanides in LaF3 using the data from Bill Carnall's 1989 paper.
       ";
4981  LoadCarnall[] := (
4982    If[ValueQ[Carnall], Return[]];
4983    carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
4984    If[!FileExistsQ[carnallFname],
4985      (PrintTemporary[">> Carnall.m not found, generating ..."];
4986        Carnall = ParseCarnall[];
4987      ),
4988      Carnall = Import[carnallFname];
4989    ];
4990  );
4991
4992  LoadChenDeltas::usage = "LoadChenDeltas[] loads the differences
       noted by Chen.";
4993  LoadChenDeltas[] := (
4994    If[ValueQ[chenDeltas], Return[]];
4995    PrintTemporary["Loading the association of discrepancies found by
        Chen ..."];
4996    chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas.m"
       }];
4997    If[!FileExistsQ[chenDeltasFname],
4998      (PrintTemporary[">> chenDeltas.m not found, generating ..."];
4999        chenDeltas = ParseChenDeltas[];
5000      ),
5001      chenDeltas = Import[chenDeltasFname];
5002    ];
5003  );
5004
5005  ParseChenDeltas::usage = "ParseChenDeltas[] parses the data found
       in ./data/the-chen-deltas-A.csv and ./data/the-chen-deltas-B.csv.
       If the option \"Export\" is set to True (True is the default),
       then the parsed data is saved to ./data/chenDeltas.m";
5006  Options[ParseChenDeltas] = {"Export" -> True};
5007  ParseChenDeltas[OptionsPattern[]] := (
5008    chenDeltasRaw = Import[FileNameJoin[{moduleDir, "data", "the-chen
       -deltas-A.csv"}]];
5009    chenDeltasRaw = chenDeltasRaw[[2 ;;]];
5010    chenDeltas = <||>;
5011    chenDeltasA = <||>;
5012    Off[Power::infy];
5013    Do[
5014      ({right, wrong} = {chenDeltasRaw[[row]][[4 ;;]],
5015        chenDeltasRaw[[row + 1]][[4 ;;]]};
5016      key = chenDeltasRaw[[row]][[1 ;; 3]];
5017      repRule = (#[[1]] -> #[[2]]*#[[1]]) & /@
5018        Transpose[{{M0, M2, M4, P2, P4, P6}, right/wrong}];
5019      chenDeltasA[key] = <|"right" -> right, "wrong" -> wrong,
5020        "repRule" -> repRule|>;
5021      chenDeltasA[{key[[1]], key[[3]], key[[2]]}] = <|"right" ->
       right,
5022        "wrong" -> wrong, "repRule" -> repRule|>;
5023      ),
5024      {row, 1, Length[chenDeltasRaw], 2}];
5025    chenDeltas["A"] = chenDeltasA;
5026
5027    chenDeltasRawB = Import[FileNameJoin[{moduleDir, "data", "the-
       chen-deltas-B.csv"}], "Text"];
5028    chenDeltasB = StringSplit[chenDeltasRawB, "\n"];
```

```
    chenDeltasB = StringSplit[#, ","] & /@ chenDeltasB;
    chenDeltasB = {ToExpression[StringTake[#[[1]], {2}]], #[[2]],
   #[[3]]} & /@ chenDeltasB;
    chenDeltas["B"] = chenDeltasB;
    On[Power::infy];
    If[OptionValue["Export"],
      (chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas
   .m"}];
      Export[chenDeltasFname, chenDeltas];
      )
      ];
    Return[chenDeltas];
  );

  ParseCarnall::usage = "ParseCarnall[] parses the data found in ./
    data/Carnall.xls. If the option \"Export\" is set to True (True is
     the default), then the parsed data is saved to ./data/Carnall.
    This data is from the tables and appendices of Carnall et al
    (1989).";
  Options[ParseCarnall] = {"Export" -> True};
  ParseCarnall[OptionsPattern[]] := (
    ions         = {"Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho"
   ,"Er","Tm","Yb"};
    templates    = StringTemplate/@StringSplit["appendix:`ion`:
    Association appendix:`ion`:Calculated appendix:`ion`:RawTable
    appendix:`ion`:Headings"," "];

    (* How many unique eigenvalues, after removing Kramer's
   degeneracy *)
    fullSizes    = AssociationThread[ions, {7, 91, 182, 1001, 1001,
   3003, 1716, 3003, 1001, 1001, 182, 91, 7}];
    carnall      = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
   "}]][[2]];
    carnallErr   = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
   "}]][[3]];

    elementNames = carnall[[1]][[2;;]];
    carnall      = carnall[[2;;]];
    carnallErr   = carnallErr[[2;;]];
    carnall      = Transpose[carnall];
    carnallErr   = Transpose[carnallErr];
    paramNames   = ToExpression/@carnall[[1]][[1;;]];
    carnall      = carnall[[2;;]];
    carnallErr   = carnallErr[[2;;]];
    carnallData  = Table[(
                    data         = carnall[[i]];
                    data         = (#[[1]]->#[[2]])&/@Select[
   Transpose[{paramNames,data}],#[[2]]!=""&];
                   elementNames[[i]]->data
                   ),
                   {i,1,13}
                   ];
    carnallData  = Association[carnallData];
    carnallNotes = Table[(
                   data        = carnallErr[[i]];
                   elementName = elementNames[[i]];
                   dataFun     = (
                       #[[1]] -> If[#[[2]]==[],
                       "Not allowed to vary in fitting.",
                       If[#[[2]]==[R],
                           "Ratio constrained by: " <> <|"Eu"->"F4/
   F2=0.713; F6/F2=0.512",
                               "Gd"->"F4/F2=0.710]",
                               "Tb"->"F4/F2=0.707"|>[elementName],
                           If[#[[2]]=="i",
                               "Interpolated",
                               #[[2]]
                           ]
                       ]
                       ]) &;
                   data = dataFun /@ Select[Transpose[{paramNames,
   data}],#[[2]]!=""&];
                   elementName->data
                     ),
                {i,1,13}
                ];
```

```
    carnallNotes  = Association[carnallNotes];

    annotatedData = Table[
                    If[NumberQ[#[[1]]],Tooltip[#[[1]],#[[2]]],""] & /@ Transpose[{paramNames/.carnallData[element],
                      paramNames/.carnallNotes[element]
                        }],
                    {element,elementNames}
                    ];
    annotatedData = Transpose[annotatedData];

    Carnall = <|"data"       -> carnallData,
        "annotations"        -> carnallNotes,
        "paramSymbols"       -> paramNames,
        "elementNames"       -> elementNames,
        "rawData"            -> carnall,
        "rawAnnotations"     -> carnallErr,
        "includedTableIons" -> ions,
        "annnotatedData"     -> annotatedData
    |>;

    Do[(
        carnallData   = Import[FileNameJoin[{moduleDir,"data","Carnall.xls"}]][[sheetIdx]];
        headers       = carnallData[[1]];
        calcIndex     = Position[headers,"Calc (1/cm)"][[1,1]];
        headers       = headers[[2;;]];
        carnallLabels = carnallData[[1]];
        carnallData   = carnallData[[2;;]];
        carnallTerms  = DeleteDuplicates[First/@carnallData];
        parsedData    = Table[(
                          rows = Select[carnallData,#[[1]]==term&];
                          rows = #[[2;;]]&/@rows;
                          rows = Transpose[rows];
                          rows = Transpose[{headers,rows}];
                          rows = Association[(#[[1]]->#[[2]])&/@rows];
                          term->rows
                          ),
                        {term,carnallTerms}
                        ];
        carnallAssoc         = Association[parsedData];
        carnallCalcEnergies = #[[calcIndex]]&/@carnallData;
        carnallCalcEnergies = If[NumberQ[#],#,Missing[]]&/@carnallCalcEnergies;
        ion                  = ions[[sheetIdx-3]];
        carnallCalcEnergies = PadRight[carnallCalcEnergies, fullSizes[ion], Missing[]];
        keys                 = #[<|"ion"->ion|>]&/@templates;
        Carnall[keys[[1]]]   = carnallAssoc;
        Carnall[keys[[2]]]   = carnallCalcEnergies;
        Carnall[keys[[3]]]   = carnallData;
        Carnall[keys[[4]]]   = headers;
        ),
    {sheetIdx,4,16}
    ];

    goodions = Select[ions,#!="Pm"&];
    expData  = Select[Transpose[Carnall["appendix:"<>#<>":RawTable"]][[1+Position[Carnall["appendix:"<>#<>":Headings"],"Exp (1/cm)"][[1,1]]]],NumberQ]&/@goodions;
    Carnall["All Experimental Data"]=AssociationThread[goodions,expData];
    If[OptionValue["Export"],
      (
        carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
        Echo["Exporting to " <> carnallFname];
        Export[carnallFname, Carnall];
      )
      ];
    Return[Carnall];
  );

  CFP::usage = "CFP[{n, NKSL}] provides a list whose first element
    echoes NKSL and whose other elements are lists with two elements
```

```
  the first one being the symbol of a parent term and the second
  being the corresponding coefficient of fractional parentage. n
  must satisfy 1 <= n <= 7.
These are according to the tables from Nielson & Koster.";

CFPAssoc::usage = "CFPAssoc is an association where keys are of
  lists of the form {num_electrons, daughterTerm, parentTerm} and
  values are the corresponding coefficients of fractional parentage.
   The terms given in string-spectroscopic notation. If a certain
  daughter term does not have a parent term, the value is 0. Loaded
  using LoadCFP[].
These are according to the tables from Nielson & Koster.";

LoadCFP::usage = "LoadCFP[] loads CFP, CFPAssoc, and CFPTable into
  the session.";
LoadCFP[] := (
  If[And[ValueQ[CFP], ValueQ[CFPTable], ValueQ[CFPAssoc]],Return
  []];

  PrintTemporary["Loading CFPtable ..."];
  CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"}];
  If[!FileExistsQ[CFPTablefname],
    (PrintTemporary[">> CFPTable.m not found, generating ..."];
      CFPTable = GenerateCFPTable["Export"->True];
    ),
    CFPTable = Import[CFPTablefname];
  ];

  PrintTemporary["Loading CFPs.m ..."];
  CFPfname = FileNameJoin[{moduleDir, "data", "CFPs.m"}];
  If[!FileExistsQ[CFPfname],
    (PrintTemporary[">> CFPs.m not found, generating ..."];
      CFP = GenerateCFP["Export"->True];
    ),
    CFP = Import[CFPfname];
  ];

  PrintTemporary["Loading CFPAssoc.m ..."];
  CFPAfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"}];
  If[!FileExistsQ[CFPAfname],
    (PrintTemporary[">> CFPAssoc.m not found, generating ..."];
      CFPAssoc = GenerateCFPAssoc["Export"->True];
    ),
    CFPAssoc = Import[CFPAfname];
  ];
);

ReducedUkTable::usage = "ReducedUkTable[{n, l = 3, SL, SpLp, k}]
  provides reduced matrix elements of the unit spherical tensor
  operator Uk. See TASS section 11-9 \"Unit Tensor Operators\".
  Loaded using LoadUk[].";

LoadUk::usage = "LoadUk[] loads into session the reduced matrix
  elements for unit tensor operators.";
LoadUk[] := (
  If[ValueQ[ReducedUkTable], Return[]];
  PrintTemporary["Loading the association of reduced matrix
  elements for unit tensor operators ..."];
  ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
  ReducedUkTable.m"}];
  If[!FileExistsQ[ReducedUkTableFname],
    (PrintTemporary[">> ReducedUkTable.m not found, generating ..."
  ];
      ReducedUkTable = GenerateReducedUkTable[7];
    ),
    ReducedUkTable = Import[ReducedUkTableFname];
  ];
);

ElectrostaticTable::usage = "ElectrostaticTable[{n, SL, SpLp}]
  provides the calculated result of Electrostatic[{n, SL, SpLp}].
  Load using LoadElectrostatic[].";

LoadElectrostatic::usage = "LoadElectrostatic[] loads the reduced
  matrix elements for the electrostatic interaction.";
LoadElectrostatic[] := (
```

```
5211      If[ValueQ[ElectrostaticTable], Return[]];
5212      PrintTemporary["Loading the association of matrix elements for
          the electrostatic interaction ..."];
5213      ElectrostaticTablefname = FileNameJoin[{moduleDir, "data", "
          ElectrostaticTable.m"}];
5214      If[!FileExistsQ[ElectrostaticTablefname],
5215        (PrintTemporary[">> ElectrostaticTable.m not found, generating
          ..."];
5216          ElectrostaticTable = GenerateElectrostaticTable[7];
5217        ),
5218        ElectrostaticTable = Import[ElectrostaticTablefname];
5219      ];
5220    );
5221
5222    LoadV1k::usage = "LoadV1k[] loads into session the matrix elements
          of V1k.";
5223    LoadV1k[] := (
5224      If[ValueQ[ReducedV1kTable], Return[]];
5225      PrintTemporary["Loading the association of matrix elements for
          V1k ..."];
5226      ReducedV1kTableFname = FileNameJoin[{moduleDir, "data", "
          ReducedV1kTable.m"}];
5227      If[!FileExistsQ[ReducedV1kTableFname],
5228        (PrintTemporary[">> ReducedV1kTable.m not found, generating ...
          "];
5229          ReducedV1kTable = GenerateReducedV1kTable[7];
5230        ),
5231        ReducedV1kTable = Import[ReducedV1kTableFname];
5232      ]
5233    );
5234
5235    LoadSpinOrbit::usage = "LoadSpinOrbit[] loads into session the
          matrix elements of the spin-orbit interaction.";
5236    LoadSpinOrbit[] := (
5237      If[ValueQ[SpinOrbitTable], Return[]];
5238      PrintTemporary["Loading the association of matrix elements for
          spin-orbit ..."];
5239      SpinOrbitTableFname = FileNameJoin[{moduleDir, "data", "
          SpinOrbitTable.m"}];
5240      If[!FileExistsQ[SpinOrbitTableFname],
5241        (
5242          PrintTemporary[">> SpinOrbitTable.m not found, generating ...
          "];
5243          SpinOrbitTable = GenerateSpinOrbitTable[7, "Export" -> True];
5244        ),
5245        SpinOrbitTable = Import[SpinOrbitTableFname];
5246      ]
5247    );
5248
5249    LoadSOOandECSOLS::usage = "LoadSOOandECSOLS[] loads into session
          the LS reduced matrix elements of the SOO-ECSO interaction.";
5250    LoadSOOandECSOLS[] := (
5251      If[ValueQ[SOOandECSOLSTable], Return[]];
5252      PrintTemporary["Loading the association of LS reduced matrix
          elements for SOO-ECSO ..."];
5253      SOOandECSOLSTableFname = FileNameJoin[{moduleDir, "data", "
          ReducedSOOandECSOLSTable.m"}];
5254      If[!FileExistsQ[SOOandECSOLSTableFname],
5255        (PrintTemporary[">> ReducedSOOandECSOLSTable.m not found,
          generating ..."];
5256          SOOandECSOLSTable = GenerateSOOandECSOLSTable[7];
5257        ),
5258        SOOandECSOLSTable = Import[SOOandECSOLSTableFname];
5259      ];
5260    );
5261
5262    LoadSOOandECSO::usage = "LoadSOOandECSO[] loads into session the
          LSJ reduced matrix elements of spin-other-orbit and
          electrostatically-correlated-spin-orbit.";
5263    LoadSOOandECSO[] := (
5264      If[ValueQ[SOOandECSOTableFname], Return[]];
5265      PrintTemporary["Loading the association of matrix elements for
          spin-other-orbit and electrostatically-correlated-spin-orbit ..."
          ];
5266      SOOandECSOTableFname = FileNameJoin[{moduleDir, "data", "
          SOOandECSOTable.m"}];
```

```
    If[!FileExistsQ[SOOandECSOTableFname],
      (PrintTemporary[">> SOOandECSOTable.m not found, generating ..."];
        SOOandECSOTable = GenerateSOOandECSOTable[7, "Export"->True];
      ),
      SOOandECSOTable = Import[SOOandECSOTableFname];
    ];
  );

  LoadT22::usage = "LoadT22[] loads into session the matrix elements of the double tensor operator T22.";
  LoadT22[] := (
    If[ValueQ[T22Table], Return[]];
    PrintTemporary["Loading the association of reduced T22 matrix elements ..."];
    T22TableFname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"}];
    If[!FileExistsQ[T22TableFname],
      (PrintTemporary[">> ReducedT22Table.m not found, generating ..."];
        T22Table = GenerateT22Table[7];
      ),
      T22Table = Import[T22TableFname];
    ];
  );

  LoadSpinSpin::usage = "LoadSpinSpin[] loads into session the matrix elements of spin-spin.";
  LoadSpinSpin[] := (
    If[ValueQ[SpinSpinTable], Return[]];
    PrintTemporary["Loading the association of matrix elements for spin-spin ..."];
    SpinSpinTableFname = FileNameJoin[{moduleDir, "data", "SpinSpinTable.m"}];
    If[!FileExistsQ[SpinSpinTableFname],
      (PrintTemporary[">> SpinSpinTable.m not found, generating ..."];
        SpinSpinTable = GenerateSpinSpinTable[7, "Export" -> True];
      ),
      SpinSpinTable = Import[SpinSpinTableFname];
    ];
  );

  LoadThreeBody::usage = "LoadThreeBody[] loads into session the matrix elements of three-body configuration-interaction effects.";
  LoadThreeBody[] := (
    If[ValueQ[ThreeBodyTable], Return[]];
    PrintTemporary["Loading the association of matrix elements for three-body configuration-interaction effects ..."];
    ThreeBodyFname   = FileNameJoin[{moduleDir, "data", "ThreeBodyTable.m"}];
    ThreeBodiesFname = FileNameJoin[{moduleDir, "data", "ThreeBodyTables.m"}];
    If[!FileExistsQ[ThreeBodyFname],
      (PrintTemporary[">> ThreeBodyTable.m not found, generating ..."];
        {ThreeBodyTable, ThreeBodyTables} = GenerateThreeBodyTables["Export" -> True];
      ),
      ThreeBodyTable  = Import[ThreeBodyFname];
      ThreeBodyTables = Import[ThreeBodiesFname];
    ];
  );

  (* ##################### Load Functions ##################### *)
  (* ############################################################ *)

End[];

LoadTermLabels[];
LoadCFP[];

EndPackage[];
```

18.2 fittings.m

This file has code useful for fitting the Hamiltonian.

```
(*-------------------------------------------------------------------*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
(*~~~+-----------------------------------------------------------+~~~*)
(*~~~|                                                           |~~~*)
(*~~~|                 _____ __  __  _                           |~~~*)
(*~~~|               / __(_) /_/ /_(_)___  ____ ______          |~~~*)
(*~~~|              / /_/ / __/ __/ / __ \/ __ `/ ___/          |~~~*)
(*~~~|             / __/ / /_/ /_/ / / / / /_/ (__  )           |~~~*)
(*~~~|            /_/ /_/\__/\__/_/_/ /_/\__, /____/            |~~~*)
(*~~~|                                   /____/                  |~~~*)
(*~~~|                                                           |~~~*)
(*~~~+-----------------------------------------------------------+~~~*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
(*~~~~+---------------------------------------------------------+~~~~*)
(*~~~~|                                                         |~~~~*)
(*~~~~|                                                         |~~~~*)
(*~~~~| This script puts together some code useful for fitting  |~~~~*)
(*~~~~|            the model Hamiltonian to data.               |~~~~*)
(*~~~~|                                                         |~~~~*)
(*~~~~|                                                         |~~~~*)
(*~~~~+---------------------------------------------------------+~~~~*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
(*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*)
+-------------------------------------------------------------------*)

BeginPackage["fittings`"];

Get[FileNameJoin[{DirectoryName[$InputFileName],"qlanth.m"}]];
Get[FileNameJoin[{DirectoryName[$InputFileName],"qonstants.m"}]];
Get[FileNameJoin[{DirectoryName[$InputFileName],"misc.m"}]];

LoadCarnall[];
LoadFreeIon[];

caseConstraints::usage="This Association contains the constraints
    that are not the same across all of the lanthanides. For instance,
     since the ratio between M2 and M0 is assumed the same for all the
     trivalent lanthanides, that one is not included here.
This association has keys equal to symbols of lanthanides and values
    equal to lists of rules that express either a parameter being held
     fixed or made proportional to another.
In Table I of Carnall 1989 these correspond to cases were values are
    given in square brackets.";
caseConstraints = <|
"Ce" -> {
    B02 -> -218.,
    B04 -> 738.,
    B06 -> 679.,
    B22 -> -50.,
    B24 -> 431.,
    B26 -> -921.,
    B44 -> 616.,
    B46 -> -348.,
    B66 -> -788.
    },
"Pr" -> {},
"Nd" -> {},
"Sm" -> {
    B22 -> -50.,
    T2 -> 300.,
    T3 -> 36.,
    T4 -> 56.,
    γ -> 1500.
    },
"Eu" -> {
    F4 -> 0.713 F2,
    F6 -> 0.512 F2,
    B22 -> -50.,
    B24 -> 597.,
    B26 -> -706.,
    B44 -> 408.,
    B46 -> -508.,
```

```
    B66 -> -692.,
    M0 -> 2.1,
    P2 -> 360.,
    T2 -> 300.,
    T3 -> 40.,
    T4 -> 60.,
    T6 -> -300.,
    T7 -> 370.,
    T8 -> 320.,
    α -> 20.16,
    β  -> -566.9,
    γ -> 1500.
    },
"Pm" -> {
    B02 -> -245.,
    B04 -> 470.,
    B06 -> 640.,
    B22 -> -50.,
    B24 -> 525.,
    B26 -> -750.,
    B44 -> 490.,
    B46 -> -450.,
    B66 -> -760.,
    F2 -> 76400.,
    F4 -> 54900.,
    F6 -> 37700.,
    M0 -> 2.4,
    P2 -> 275.,
    T2 -> 300.,
    T3 -> 35.,
    T4 -> 58.,
    T6 -> -310.,
    T7 -> 350.,
    T8 -> 320.,
    α -> 20.5,
    β -> -560.,
    γ -> 1475.,
    ζ -> 1025.},
"Gd" -> {
    F4 -> 0.710 F2,
    B02 -> -231.,
    B04 -> 604.,
    B06 -> 280.,
    B22 -> -99.,
    B24 -> 340.,
    B26 -> -721.,
    B44 -> 452.,
    B46 -> -204.,
    B66 -> -509.,
    T2 -> 300.,
    T3 -> 42.,
    T4 -> 62.,
    T6 -> -295.,
    T7 -> 350.,
    T8 -> 310.,
    β -> -600.,
    γ -> 1575.
    },
"Tb" -> {
    F4 -> 0.707 F2,
    T2 -> 320.,
    T3 -> 40.,
    T4 -> 50.,
    γ -> 1650.
    },
"Dy" -> {},
"Ho" -> {
    B02 -> -240.,
    T2 -> 400.,
    γ -> 1800.
    },
"Er" -> {
    T2 -> 400.,
    γ -> 1800.
    },
"Tm" -> {
```

```
    T2 -> 400.,
    γ -> 1820.
    },
"Yb" -> {
    B02 -> -249.,
    B04 -> 457.,
    B06 -> 282.,
    B22 -> -105.,
    B24 -> 320.,
    B26 -> -482.,
    B44 -> 428.,
    B46 -> -234.,
    B66 -> -492.
}
|>;

variedSymbols =<|
    "Ce" -> {ζ},
    "Pr" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             α, β, γ,
             ζ},
    "Nd" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             T2, T3, T4, T6, T7, T8,
             α, β, γ,
             ζ},
    "Pm" -> {},
    "Sm" -> {B02, B04, B06, B24, B26, B44, B46, B66,
             F2, F4, F6, M0, P2,
             T6, T7, T8,
             α, β, ζ},
    "Eu" -> {B02, B04, B06,
             F2, F4, F6, ζ},
    "Gd" -> {F2, F4, F6,
             M0, P2,
             α, ζ},
    "Tb" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             T6, T7, T8,
             α, β, ζ},
    "Dy" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             T2, T3, T4, T6, T7, T8,
             α, β, γ, ζ},
    "Ho" -> {B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             T3, T4, T6, T7, T8,
             α, β, ζ},
    "Er" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             T3, T4, T6, T7, T8, α, β, ζ},
    "Tm" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             F2, F4, F6,
             M0, P2,
             α, β, ζ},
    "Yb" -> {ζ}
    |>;

caseConstraintsM0::usage="This association contains the symbols that
    are held constant in fitting different ions. It only contains
    symbols for which the constraint doesn't apply to all ions.
This association has keys equal to symbols of lanthanides and values
    equal to associations with the symbols for the parameters that are
     held fixed or made proportional to another. If the value is to be
     held constant a placeholder value of 0 is given, if a ratio
    constraint is given, the value is constraint.
The operator basis for which this is applicable is the mostly-
    orthogonal basis.
";
```

```
caseConstraintsM0 = <|
"Ce"-> {
    B02 -> 0,
    B04 -> 0,
    B06 -> 0,
    B22 -> 0,
    B24 -> 0,
    B26 -> 0,
    B44 -> 0,
    B46 -> 0,
    B66 -> 0
    },
"Pr"-> {},
"Nd"-> {},
"Sm"-> {
    B22 -> 0,
    T2p -> 0,
    T3 -> 0,
    T4 -> 0,
    γp -> 0
    },
"Eu"-> {
    E2p -> 0.0049 E1p,
    E3p -> 0.098 E1p,
    B22 -> 0,
    B24 -> 0,
    B26 -> 0,
    B44 -> 0,
    B46 -> 0,
    B66 -> 0,
    M0  -> 0,
    P2  -> 0,
    T2p -> 0,
    T3  -> 0,
    T4  -> 0,
    T6  -> 0,
    T7  -> 0,
    T8  -> 0,
    αp -> 0,
    βp -> 0,
    γp -> 0
    },
"Pm"-> {
    B02 -> 0,
    B04 -> 0,
    B06 -> 0,
    B22 -> 0,
    B24 -> 0,
    B26 -> 0,
    B44 -> 0,
    B46 -> 0,
    B66 -> 0,
    E1p -> 0,
    E2p -> 0,
    E3p -> 0,
    M0 -> 0,
    P2 -> 0,
    T2p -> 0,
    T3 -> 0,
    T4 -> 0,
    T6 -> 0,
    T7 -> 0,
    T8 -> 0,
    αp -> 0,
    βp -> 0,
    γp -> 0,
    ζ -> 0
    },
"Gd"-> {
    E2p -> 0.0049 E1p,
    B02 -> 0,
    B04 -> 0,
    B06 -> 0,
    B22 -> 0,
    B24 -> 0,
    B26 -> 0,
```

```
    B44 -> 0,
    B46 -> 0,
    B66 -> 0,
    T2p -> 0,
    T3 -> 0,
    T4 -> 0,
    T6 -> 0,
    T7 -> 0,
    T8 -> 0,
    βp -> 0,
    γp -> 0
    },
"Tb"-> {
    E2p -> 0.0049 E1p,
    T2p -> 0,
    T3 -> 0,
    T4 -> 0,
    γp -> 0
    },
"Dy"-> {},
"Ho"-> {
    B02 -> 0,
    T2p -> 0,
    γp -> 0
    },
"Er"-> {
    T2p -> 0,
    γp -> 0
    },
"Tm"-> {
    γp -> 0
    },
"Yb"-> {
    B02 -> 0,
    B04 -> 0,
    B06 -> 0,
    B22 -> 0,
    B24 -> 0,
    B26 -> 0,
    B44 -> 0,
    B46 -> 0,
    B66 -> 0
}
|>;

variedSymbolsM0 =<|
    "Ce" -> {ζ},
    "Pr" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             E1p, E2p, E3p,
             M0, P2,
             αp, βp, γp,
             ζ},
    "Nd" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             E1p, E2p, E3p,
             M0, P2,
             T2p, T3, T4, T6, T7, T8,
             αp, βp, γp,
             ζ},
    "Pm" -> {},
    "Sm" -> {B02, B04, B06, B24, B26, B44, B46, B66,
             E1p, E2p, E3p, M0, P2,
             T6, T7, T8,
             αp, βp, ζ},
    "Eu" -> {B02, B04, B06,
             E1p, E2p, E3p, ζ},
    "Gd" -> {E1p, E2p, E3p,
             M0, P2,
             αp, ζ},
    "Tb" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             E1p, E2p, E3p,
             M0, P2,
             T6, T7, T8,
             αp, βp, ζ},
    "Dy" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
             E1p, E2p, E3p,
             M0, P2,
```

```
            T2p, T3, T4, T6, T7, T8,
            αp, βp, γp, ζ},
    "Ho" -> {B04, B06, B22, B24, B26, B44, B46, B66,
            E1p, E2p, E3p,
            M0, P2,
            T3, T4, T6, T7, T8,
            αp, βp, ζ},
    "Er" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
            E1p, E2p, E3p,
            M0, P2,
            T3, T4, T6, T7, T8, αp, βp, ζ},
    "Tm" -> {B02, B04, B06, B22, B24, B26, B44, B46, B66,
            E1p, E2p, E3p,
            M0, P2,
            αp, βp, ζ},
    "Yb" -> {ζ}
    |>;

caseConstraintsLiYF4 = <|
  "Ce" -> {
    B04 -> -1043.,
    B44 -> -1249.,
    B06 -> -65.,
    B46 -> -1069.
    },
  "Pr" -> {
    β -> -644.,
    γ -> 1413.,
    M0 -> 1.88,
    P2 -> 244.
    },
  "Nd" -> {
    M0 -> 1.85
    },
  "Sm" -> {
    α -> 20.5,
    β -> -616.,
    γ -> 1565.,
    T2 -> 282.,
    T3 -> 26.,
    T4 -> 71.,
    T6 -> -257.,
    T7 -> 314.,
    T8 -> 328.,
    M0 -> 2.38,
    P2 -> 336.
    },
  "Eu" -> {
    T2 -> 370.,
    T3 -> 40.,
    T4 -> 40.,
    T6 -> -300.,
    T7 -> 380.,
    T8 -> 370.
    },
  "Tb" -> {
    F4 -> 0.709 F2,
    F6 -> 0.503 F2,
    α -> 17.6,
    β -> -581.,
    γ -> 1792.,
    T2 -> 330.,
    T3 -> 40.,
    T4 -> 45.,
    T6 -> -365.,
    T7 -> 320.,
    T8 -> 349.,
    M0 -> 2.7,
    P2 -> 482.
    },
  "Dy" -> {
    (* F4 -> 0.707 F2,
    F6 -> 0.516 F2, *)
    F2 -> 90421,
    F4 -> 63928,
    F6 -> 46657,
```

```
    α -> 17.9,
    β -> -628.,
    γ -> 1790.,
    T2 -> 326.,
    T3 -> 23.,
    T4 -> 83.,
    T6 -> -294.,
    T7 -> 403.,
    T8 -> 340.,
    M0 -> 4.46,
    P2 -> 610.,
    B46 -> -700.
    },
  "Ho" -> {
    α -> 17.2,
    β -> -596.,
    γ -> 1839.,
    T2 -> 365.,
    T3 -> 37.,
    T4 -> 95.,
    T6 -> -274.,
    T7 -> 331.,
    T8 -> 343.,
    P2 -> 582.
    },
  "Er" -> {},
  "Tm" -> {
    α -> 17.3,
    β -> -665.,
    γ -> 1936.,
    M0 -> 4.93,
    P2 -> 730.,
    T2 -> 400.
    },
  "Yb" -> {
    B06 -> -23.,
    B46 -> -512.
    }
  |>;

variedSymbolsLiYF4 = <|
  "Ce" -> {
    B02, ζ
    },
  "Pr" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    α, ζ
    },
  "Nd" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    P2,
    T2, T3, T4, T6, T7, T8,
    α, β, γ, ζ
    },
  "Sm" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    ζ
    },
  "Eu" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    M0, P2,
    α, β, γ, ζ
    },
  "Tb" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    ζ
    },
  "Dy" -> {
    B02, B04, B06, B44,
    F2, F4, F6,
    ζ
```

```
    },
  "Ho" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    M0,
    ζ
    },
  "Er" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    M0, P2,
    T2, T3, T4, T6, T7,  T8,
    α, β, γ,
    ζ
    },
  "Tm" -> {
    B02, B04, B06, B44, B46,
    F2, F4, F6,
    ζ
    },
  "Yb" -> {
    B02, B04, B44,
    ζ
    }
  |>;

paramsChengLiYF4::usage="This association has the model parameters as fitted by Cheng et. al \"Crystal-field analyses for trivalent lanthanide ions in LiYF4\".";
paramsChengLiYF4 = <|
  "Ce" -> {
    ζ -> 630.,
    B02 -> 354., B04 -> -1043.,
    B44 -> -1249., B06 -> -65.,
    B46 -> -1069.
    },
  "Pr" -> {
    F2 -> 68955., F4 -> 50505., F6 -> 33098.,
    ζ -> 748.,
    α -> 23.3, β -> -644., γ -> 1413.,
    M0 -> 1.88, P2 -> 244.,
    B02 -> 512.,   B04 -> -1127.,
    B44 -> -1239., B06 -> -85.,
    B46 -> -1205.
    },
 "Nd" -> {
    F2 -> 72952., F4 -> 52681., F6 -> 35476.,
    ζ -> 877.,
    α -> 21., β -> -579., γ -> 1446.,
    T2 -> 210., T3 -> 41., T4 -> 74., T6 -> -293., T7 -> 321., T8 -> 205.,
    M0 -> 1.85, P2 -> 304.,
    B02 -> 391.,   B04 -> -1031.,
    B44 -> -1271., B06 -> -28.,
    B46 -> -1046.
    },
 "Sm" -> {
    F2 -> 79515., F4 -> 56766., F6 -> 40078.,
    ζ -> 1168.,
    α -> 20.5, β -> -616., γ -> 1565.,
    T2 -> 282., T3 -> 26., T4 -> 71., T6 -> -257., T7 -> 314., T8 -> 328.,
    M0 -> 2.38, P2 -> 336.,
    B02 -> 370.,  B04 -> -757.,
    B44 -> -941., B06 -> -67.,
    B46 -> -895.
    },
 "Eu" -> {
    F2 -> 82573., F4 -> 59646., F6 -> 43203.,
    ζ -> 1329.,
    α -> 21.6, β -> -482., γ -> 1140.,
    T2 -> 370., T3 -> 40., T4 -> 40., T6 -> -300., T7 -> 380., T8 -> 370.,
    M0 -> 2.41, P2 -> 332.,
    B02 -> 339.,   B04 -> -733.,
    B44 -> -1067., B06 -> -36.,
```

```
    B46 -> -764.
    },
 "Tb" -> {
    F2 -> 90972., F4 -> 64499., F6 -> 45759.,
    ζ -> 1702.,
    α -> 17.6, β -> -581., γ -> 1792.,
    T2 -> 330., T3 -> 40., T4 -> 45., T6 -> -365., T7 -> 320., T8 ->
    349.,
    M0 -> 2.7, P2 -> 482.,
    B02 -> 413.,   B04 -> -867.,
    B44 -> -1114., B06 -> -41.,
    B46 -> -736.
    },
 "Dy" -> {
    F0 -> 0,
    F2 -> 90421., F4 -> 63928., F6 -> 46657.,
    ζ -> 1895.,
    α -> 17.9, β -> -628., γ -> 1790.,
    T2 -> 326., T3 -> 23., T4 -> 83., T6 -> -294., T7 -> 403., T8 ->
    340.,
    M0 -> 4.46, P2 -> 610.,
    B02 -> 360.,  B04 -> -737.,
    B44 -> -943., B06 -> -35.,
    B46 -> -700.
    },
 "Ho" -> {
    F2 -> 93512., F4 -> 66084., F6 -> 49765.,
    ζ -> 2126.,
    α -> 17.2, β -> -596., γ -> 1839.,
    T2 -> 365., T3 -> 37., T4 -> 95., T6 -> -274., T7 -> 331., T8 ->
    343.,
    M0 -> 3.92, P2 -> 582.,
    B02 -> 386.,  B04 -> -629.,
    B44 -> -841., B06 -> -33.,
    B46 -> -687.
    },
 "Er" -> {
    F2 -> 97326., F4 -> 67987., F6 -> 53651.,
    ζ -> 2377.,
    α -> 18.1, β -> -599., γ -> 1870.,
    T2 -> 380., T3 -> 41., T4 -> 69., T6 -> -356., T7 -> 239., T8 ->
    390.,
    M0 -> 4.41, P2 -> 795.,
    B02 -> 325.,   B04 -> -749.,
    B44 -> -1014., B06 -> -19.,
    B46 -> -635.
    },
 "Tm" -> {
    F0 -> 0.,
    T2 -> 0.,
    F2 -> 101938., F4 -> 71553., F6 -> 51359.,
    ζ -> 2632.,
    α -> 17.3, β -> -665., γ -> 1936.,
    M0 -> 4.93, P2 -> 730.,
    B02 -> 339.,  B04 -> -627.,
    B44 -> -913., B06 -> -39.,
    B46 -> -584.
    },
 "Yb" -> {
    ζ -> 2916.,
    B02 -> 446.,  B04 -> -560.,
    B44 -> -843., B06 -> -23.,
    B46 -> -512.
    }
  |>;

Jiggle::usage = "Jiggle[num, wiggleRoom] takes a number and
    randomizes it a little by adding or subtracting a random fraction
    of itself. The fraction is controlled by wiggleRoom.";
Jiggle[num_, wiggleRoom_ : 0.1] := RandomReal[{1 - wiggleRoom, 1 +
    wiggleRoom}] * num;

AddToList::usage = "AddToList[list, element, maxSize, addOnlyNew]
    prepends the element to list and returns the list. If maxSize is
    reached, the last element is dropped. If addOnlyNew is True (the
    default), the element is only added if it is different from the
```

```
    last element.";
AddToList[list_, element_, maxSize_, addOnlyNew_ : True] := Module[{
    tempList = If[
      addOnlyNew,
      If[
       Length[list] == 0,
       {element},
       If[
        element != list[[-1]],
        Append[list, element],
        list
        ]
       ],
      Append[list, element]
      ]},
   If[Length[tempList] > maxSize,
    Drop[tempList, Length[tempList] - maxSize],
    tempList]
   ];

ProgressNotebook::usage="ProgressNotebook[] creates a progress
    notebook for the solver. This notebook includes a plot of the RMS
    history and the current parameter values. The notebook is returned
    . The RMS history and the parameter values are updated by setting
    the variables rmsHistory and paramSols. The variables
    stringPartialVars and paramSols are used to display the parameter
    values in the notebook.";
Options[ProgressNotebook] = {
    "Basic" -> True,
    "UpdateInterval" -> 0.5};
ProgressNotebook[OptionsPattern[]] := (
   nb = Which[
   OptionValue["Basic"],
   CreateDocument[(
      {
       Dynamic[
        TextCell[
         If[
          Length[paramSols] > 0,
          TableForm[
            Prepend[
              Transpose[{stringPartialVars,
              paramSols[[-1]]}],
              {"RMS",rmsHistory[[-1]]}]
              ],
          ""
          ],
         "Output"
         ],
        TrackedSymbols :> {paramSols, stringPartialVars},
        UpdateInterval -> OptionValue["UpdateInterval"]
        ]
       }
      ),
     WindowSize -> {600, 1000},
     WindowSelected -> True,
     TextAlignment -> Center,
     WindowTitle -> "Solver Progress"
     ],
   True,
   CreateDocument[(
      {
        "",
        Dynamic[Framed[progressMessage],
          UpdateInterval -> OptionValue["UpdateInterval"]],
        Dynamic[
        GraphicsColumn[
         {ListPlot[rmsHistory,
           PlotMarkers -> "OpenMarkers",
           Frame -> True,
           FrameLabel -> {"Iteration", "RMS"},
           ImageSize -> 800,
           AspectRatio -> 1/3,
           FrameStyle -> Directive[Thick, 15],
           PlotLabel -> If[Length[rmsHistory] != 0, rmsHistory[[-1]],
    ""]
```

```
            ],
           ListPlot[(#/#[[1]]) & /@ Transpose[paramSols],
            Joined -> True,
            PlotRange -> {All, {-5, 5}},
            Frame -> True,
            ImageSize -> 800,
            AspectRatio -> 1,
            FrameStyle -> Directive[Thick, 15],
            FrameLabel -> {"Iteration", "Params"}
            ]
           }
          ],
         TrackedSymbols :> {rmsHistory, paramSols},
         UpdateInterval -> OptionValue["UpdateInterval"]
         ],
        Dynamic[
         TextCell[
          If[
           Length[paramSols] > 0,
           TableForm[Transpose[{stringPartialVars, paramSols[[-1]]}]],
           ""
           ],
          "Output"
          ],
         TrackedSymbols :> {paramSols, stringPartialVars}
         ]
        }
       ),
      WindowSize -> {600, 1000},
      WindowSelected -> True,
      TextAlignment -> Center,
      WindowTitle -> "Solver Progress"
      ]
    ];
    Return[nb];
    );

energyCostFunTemplate::usage="energyCostFunTemplate is template used to define the cost function for the energy matching. The template is used to define a function TheRightEnergyPath that takes a list of variables and returns the RMS of the energy differences between the computed and the experimental energies. The template requires the values to the following keys to be provided: `vars` and `varPatterns`";
energyCostFunTemplate = StringTemplate["
TheRightEnergyPath[`varPatterns`]:= (
  {eigenEnergies, eigenVecs} = Eigensystem[compHam[`vars`]];
  ordering      = Ordering[eigenEnergies];
  eigenEnergies = eigenEnergies - Min[eigenEnergies];
  states        = Transpose[{eigenEnergies, eigenVecs}];
  states        = states[[ordering]];
  coarseStates  = ParseStates[states, basis];
  coarseStates  = {#[[1]],#[[-1]]}& /@ coarseStates;
  (* The eigenvectors need to be simplified in order to compare labels to labels *)
  missingLevels = Length[coarseStates]-Length[expData];
  (* The energies are in the first element of the tuples. *)
  energyDiffFun = (Abs[#1[[1]]-#2[[1]]])&;
  (* match disregarding labels *)
  energyFlow    = FlowMatching[coarseStates,
                  expData,
                  \"notMatched\" -> missingLevels,
                  \"CostFun\"    -> energyDiffFun
                  ];
  energyPairs   = {#[[1]][[1]], #[[2]][[1]]} & /@ energyFlow[[1]];
  energyRms     = Sqrt[Total[(Abs[#[[2]]-#[[1]]])^2 & /@ energyPairs] / Length[energyPairs]];
  Return[energyRms];
)"];

AppendToLog[message_, file_String] := Module[
  {timestamp = DateString["ISODateTime"], msgString},
  (
    msgString = ToString[message, InputForm]; (* Convert any expression to a string *)
    OpenAppend[file];
```

```
    WriteString[file, timestamp, " - ", msgString, "\n"];
    Close[file];
  )
  ];

energyAndLabelCostFunTemplate::usage="energyAndLabelCostFunTemplate
    is a template used to define the cost function that includes both
    the differences between energies and the differences between
    labels. The template is used to define a function
    TheRightSignedPath that takes a list of variables and returns the
    RMS of the energy differences between the computed and the
    experimental energies together with a term that depends on the
    differences between the labels. The template requires the values
    to the following keys to be provided: `vars` and `varPatterns`";
energyAndLabelCostFunTemplate = StringTemplate["
TheRightSignedPath[`varPatterns`] := Module[
  {energyRms, eigenEnergies, eigenVecs, ordering, states,
    coarseStates, missingLevels, energyDiffFun, energyFlow,
    energyPairs, energyAndLabelFun, energyAndLabelFlow, totalAvgCost},
  (
    {eigenEnergies, eigenVecs} = Eigensystem[compHam[`vars`]];
    ordering      = Ordering[eigenEnergies];
    eigenEnergies = eigenEnergies - Min[eigenEnergies];
    states        = Transpose[{eigenEnergies, eigenVecs}];
    states        = states[[ordering]];
    coarseStates  = ParseStates[states, basis];

    (* The eigenvectors need to be simplified in order to compare
    labels to labels *)
    coarseStates  = {#[[1]],#[[-1]]}& /@ coarseStates;
    missingLevels = Length[coarseStates]-Length[expData];

    (* The energies are in the first element of the tuples. *)
    energyDiffFun = ( Abs[#1[[1]]-#2[[1]]] ) &;

    (* matching disregarding labels to get overall scale for scaling
    differences in labels *)
    energyFlow    = FlowMatching[coarseStates,
                      expData,
                      \"notMatched\" -> missingLevels,
                      \"CostFun\"    -> energyDiffFun
                      ];
    energyPairs   = {#[[1]][[1]], #[[2]][[1]]}&/@energyFlow[[1]];
    energyRms     = Sqrt[Total[(Abs[#[[2]]-#[[1]]])^2 & /@
    energyPairs]/Length[energyPairs]];

    (* matching using both labels and energies  *)
    energyAndLabelFun  = With[{del=energyRms},
                (Abs[#1[[1]]-#2[[1]]] +
                If[#1[[2]]==#2[[2]],
                  0.,
                  del])&];

    (* energyAndLabelFun  = With[{del=energyRms},
                (Abs[#1[[1]]-#2[[1]]] +
                del*EditDistance[#1[[2]],#2[[2]]])&]; *)
    energyAndLabelFun  = ( Abs[#1[[1]] - #2[[1]]] + EditDistance
    [#1[[2]],#2[[2]]] )&;
    energyAndLabelFlow = FlowMatching[coarseStates,
                         expData,
                         \"notMatched\" -> missingLevels,
                         \"CostFun\"    -> energyAndLabelFun
                         ];
    totalAvgCost       = Total[energyAndLabelFun@@# & /@
    energyAndLabelFlow[[1]]]/Length[energyAndLabelFlow[[1]]];
    Return[totalAvgCost];
  )
]"];

Constrainer::usage = "Constrainer[problemVars, ln] returns a list of
    constraints for the variables in problemVars for trivalent
    lanthanide ion ln. problemVars are standard model symbols (F2, F4,
     ...). The ranges returned are based in the fitted parameters for
    LaF3 as found in Carnall et al. They could probably be more fine
    grained, but these ranges are seen to describe all the ions in
    that case.";
```

```
Constrainer[problemVars_, ln_] := (
  slater = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {#, (20000. < # < 120000.)} & /@ {F2, F4, F6}
    ];
  alpha = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {{α, 14. < α < 22.}}
    ];
  zeta = {{ζ, 500. < ζ < 3200.}};
  beta = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {{β, -1000. < β < -400.}}
    ];
  gamma = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {{γ, 1000. < γ < 2000.}}
    ];
  tees = Which[
    ln == "Tm",
    {100. < T2 < 500.},
    MemberQ[{"Ce", "Pr", "Yb"}, ln],
    {},
    True,
    {#, -500. < # < 500.} & /@ {T2, T3, T4, T6, T7, T8}];
  marvins = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {{M0, 1.0 < M0 < 5.0}}
    ];
  peas = Which[
    MemberQ[{"Ce", "Yb"}, ln],
    {},
    True,
    {{P2, -200. < P2 < 1200.}}
    ];
  crystalRanges = {#, (-2000. < # < 2000.)} & /@ (Intersection[
      cfSymbols, problemVars]);
  allCons =
   Join[slater, zeta, alpha, beta, gamma, tees, marvins, peas,
    crystalRanges];
  allCons = Select[allCons, MemberQ[problemVars, #[[1]]] &];
  Return[Flatten[Rest /@ allCons]]
  );

Options[LogSol] = {"PrintFun" -> PrintTemporary};
LogSol::usage = "LogSol[expr, prefix] saves the given expression to a
    file. The file is named with the given prefix and a created UUID.
    The file is saved in the \"log\" directory under the current
   directory. The file is saved in the format of a .m file. The
   function returns the name of the file.";
LogSol[theSolution_, prefix_, OptionsPattern[]] := (
  PrintFun = OptionValue["PrintFun"];
  fname = prefix <> "-sols-" <> CreateUUID[] <> ".m";
  fname = FileNameJoin[{".", "log", fname}];
  PrintFun["Saving solution to: ", fname];
  Export[fname, theSolution];
  Return[fname];
  );

ClassicalFit::usage="ClassicalFit[numE, expData, excludeDataIndices,
   problemVars, startValues, constraints] fits the given expData in
   an f^numE configuration, by using the symbols in problemVars. The
   symbols given in problemVars may be constrained or held constant,
   this being controlled by constraints list which is a list of
   replacement rules expressing desired constraints. The constraints
   list additional constraints imposed upon the model parameters that
```

```
    remain once other simplifications have been \"baked\" into the
    compiled Hamiltonians that are used to increase the speed of the
    calculation.

Important, note that in the case of odd number of electrons the given
    data must explicitly include the Kramers degeneracy;
    excludeDataIndices must be compatible with this.

The list expData needs to be a list of lists with the only
    restriction that the first element of them corresponds to energies
    of levels. In this list, an empty value can be used to indicate
    known gaps in the data. Even if the energy value for a level is
    known (and given in expData) certain values can be omitted from
    the fitting procedure through the list excludeDataIndices, which
    correspond to indices in expData that should be skipped over.

The Hamiltonian used for fitting is a version that has been truncated
    either by using the maximum energy given in expData or by
    manually setting a truncation energy using the option \"
    TruncationEnergy\".

The option \"Experimental Uncertainty in K\" is the estimated
    uncertainty in the differences between the calculated and the
    experimental energy levels. This is used to estimate the
    uncertainty in the fitted parameters. Admittedly this will be a
    rough estimate (at least on the contribution of the calculated
    uncertainty), but it is better than nothing and may at least
    provide a lower bound to the uncertainty in the fitted parameters.
    It is assumed that the uncertainty in the differences between the
    calculated and the experimental energy levels is the same for all
    of them.

The list startValues is a list with all of the parameters needed to
    define the Hamiltonian (including the initial values for
    problemVars).

The function saves the solution to a file. The file is named with a
    prefix (controlled by the option \"FilePrefix\") and a UUID. The
    file is saved in the log sub-directory as a .m file.

Here's a description of the different parts of this function: first
    the Hamiltonian is assembled and simplified using the given
    simplifications. Then the intermediate coupling basis is
    calculated using the free-ion parameters for the given lanthanide.
    The Hamiltonian is then changed to the intermediate coupling
    basis and truncated. The truncated Hamiltonian is then compiled
    into a function that can be used to calculate the energy levels of
    the truncated Hamiltonian. The function that calculates the
    energy levels is then used to fit the experimental data. The
    fitting is done using FindMinimum with the Levenberg-Marquardt
    method.

The function returns an association with the following keys:

- \"bestRMS\" which is the best \[Sigma] value found.
- \"bestParams\" which is the best set of parameters found for the
    variables that were not constrained.
- \"bestParamsWithConstraints\" which has the best set of parameters
    (from - \"bestParams\") together with the used constraints. These
    include all the parameters in the model, even those that were not
    fitted for.
- \"paramSols\" which is a list of the parameters trajectories during
    the stepping of the fitting algorithm.
- \"timeTaken/s\" which is the time taken to find the best fit.
- \"simplifier\" which is the simplifier used to simplify the
    Hamiltonian.
- \"excludeDataIndices\" as given in the input.
- \"startValues\" as given in the input.

- \"freeIonSymbols\" which are the symbols used in the intermediate
    coupling basis.
- \"truncationEnergy\" which is the energy used to truncate the
    Hamiltonian, if it was set to Automatic, the value here is the
    actual energy used.
- \"numE\" which is the number of electrons in the f^numE
    configuration.
```

```
939 - \"expData\" which is the experimental data used for fitting.
940 - \"problemVars\" which are the symbols considered for fitting
941
942 - \"maxIterations\" which is the maximum number of iterations used by
        NMinimize.
943 - \"hamDim\" which is the dimension of the full Hamiltonian.
944 - \"allVars\" which are all the symbols defining the Hamiltonian
      under the aggregate simplifications.
945 - \"freeBies\" which are the free-ion parameters used to define the
      intermediate coupling basis.
946 - \"truncatedDim\" which is the dimension of the truncated
      Hamiltonian.
947 - \"compiledIntermediateFname\" the file name of the compiled
      function used for the truncated Hamiltonian.
948
949 - \"fittedLevels\" which is the number of levels fitted for.
950 - \"actualSteps\" the number of steps that FindMiniminum actually
      took.
951 - \"solWithUncertainty\" which is a list of replacement rules of the
      form (paramSymbol -> {bestEstimate, uncertainty}).
952 - \"rmsHistory\" which is a list of the \[Sigma] values found during
      the fitting.
953 - \"Appendix\" which is an association appended to the log file under
        the key \"Appendix\".
954 - \"presentDataIndices\" which is the list of indices in expData that
        were used for fitting, this takes into account both the empty
      indices in expData and also the indices in excludeDataIndices.
955
956 - \"states\" which contains a list of eigenvalues and eigenvectors
      for the fitted model, this is only available if the option \"
      SaveEigenvectors\" is set to True; if a general shift of energy
      was allowed for in the fitting, then the energies are shifted
      accordingly.
957 - \"energies\" which is a list of the energies of the fitted levels,
      this is only available if the option \"SaveEigenvectors\" is set
      to False. If a general shift of energy was allowed for in the
      fitting, then the energies are shifted accordingly.
958 - \"degreesOfFreedom\" which is equal to the nunber of fitted state
      energies minus the number of parameters used in fitting.
959
960 The function admits the following options with corresponding default
      values:
961 - \"MaxHistory\" : determines how long the logs for the solver can be
      .
962 - \"MaxIterations\": determines the maximum number of iterations used
        by NMinimize.
963 - \"FilePrefix\" : the prefix to use for the subfolder in the log
      filder, in which the solution files are saved, by default this is
      \"calcs\" so that the calculation files are saved under the
      directory \"log/calcs\".
964 - \"AddConstantShift\" : if True then a constant shift is allowed in
      the fitting, default is False. If this is the case the variable \"
      \[Epsilon]\" is added to the list of variables to be fitted for,
      it must not be included in problemVars.
965
966 - \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
      5.
967 - \"TrucationEnergy\": if Automatic then the maximum energy in
      expData is taken, else it takes the value set by this option. In
      all cases the energies in expData are only considered up to this
      value.
968 - \"PrintFun\": the function used to print progress messages, the
      default is PrintTemporary.
969 - \"RefParamsVintage\": the vintage of the reference parameters to
      use. The reference parameters are both used to determine the
      truncated Hamiltonian, and also as starting values for the solver.
       It may be \"LaF3\", in which case reference parameters from
      Carnall are used. It may also be \"LiYF4\", in which case the
      reference parameters from the LiYF4 paper are used. It may also be
       Automatic, in which case the given experimental data is used to
      determine starting values for F^k and ζ. It may also be a list or
      association that provides values for the Slater integrals and spin
      -orbit coupling, the remaining necessary parameters complemented
      by using \"LaF3\".
970
971 - \"ProgressView\": whether or not a progress window will be opened
```

```
    to show the progress of the solver, the default is True.
- \"SignatureCheck\": if True then then the function returns
    prematurely, returning a list with the symbols that would have
    defined the Hamiltonian after all simplifications have been
    applied. Useful to check the entire parameter set that the
    Hamiltonian has, which has to match one-to-one what is provided by
     startValues.
- \"SaveEigenvectors\": if True then the both the eigenvectors and
    eigenvalues are saved under the \"states\" key of the returned
    association. If False then only the energies are saved, the
    default is False.

- \"AppendToLogFile\": an association appended to the log file under
    the key \"Appendix\".
- \"MagneticSimplifier\": a list of replacement rules to simplify the
     Marvin and pesudo-magnetic paramters. Here the ratios of the
    Marvin parameters and the pseudo-magnetic parameters are defined
    to simplify the magnetic part of the Hamiltonian.
- \"MagFieldSimplifier\": a list of replacement rules to specify a
    magnetic field (in T), if set to {}, then {Bx, By, Bz} can also be
     used as variables to be fitted for.

- \"SymmetrySimplifier\": a list of replacements rules to simplify
    the crystal field.
- \"OtherSimplifier\": an additional list of replacement rules that
    are applied to the Hamiltonian before computing with it. Here the
    spin-spin contribution can be turned off by setting \[Sigma]SS->0,
     which is the default.
";
Options[ClassicalFit] = {
  "MaxHistory"       -> 200,
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "calcs",
  "ProgressView"     -> True,
  "TruncationEnergy" -> Automatic,
  "AccuracyGoal"     -> 5,
  "PrintFun"         -> PrintTemporary,
  "RefParamsVintage" -> "LaF3",
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "SaveToLog"        -> False,
  "Energy Uncertainty in K" -> Automatic,
  "MagneticSimplifier" -> {
    M2 -> 56/100 M0,
    M4 -> 31/100 M0,
    P4 -> 1/2 P2,
    P6 -> 1/10 P2
    },
  "MagFieldSimplifier" -> {
    Bx -> 0,
    By -> 0,
    Bz -> 0
    },
  "SymmetrySimplifier" -> {
    B12->0, B14->0, B16->0, B34->0, B36->0, B56->0,
    S12->0 ,S14->0, S16->0, S22->0, S24->0, S26->0,
    S34->0 ,S36->0, S44->0, S46->0, S56->0, S66->0
    },
  "OtherSimplifier" -> {
    F0->0,
    P0->0,
    \[Sigma]SS->0,
    T11p->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0,
    wChErrA ->0, wChErrB->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19
    ->0,
      T2p->0},
    2 -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
```

```
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19->0,
      T2p->0
      },
    3  -> {},
    4  -> {},
    5  -> {},
    6  -> {},
    7  -> {},
    8  -> {},
    9  -> {},
    10 -> {},
    11 -> {},
    12 -> {
      T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19->0,
      T2p->0
      },
    13->{
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17->0, T19->0,
      T2p->0
      }
    |>,
    "FreeIonSymbols" -> {F0, F2, F4, F6, ζ}
};
ClassicalFit[numE_Integer, expData_List, excludeDataIndices_List, problemVars_List, startValues_Association, constraints_List, OptionsPattern[]]:=Module[
  {
    accuracyGoal,
    allFreeEnergies,
    allFreeEnergiesSorted, allVars, allVarsVec,
    argsForEvalInsideOfTheIntermediateSystems,
    argsOfTheIntermediateEigensystems, aVar, aVarPosition,
    basis, basisChanger, basisChangerBlocks,
    bestParams, bestRMS, blockShifts, blockSizes,
    compiledDiagonal, compiledIntermediateFname,
    constrainedProblemVars, constrainedProblemVarsList,
    currentRMS, degressOfFreedom, dependentVars,
    diagonalBlocks, diagonalScalarBlocks, diff,
    eigenEnergies, eigenvalueDispenserTemplate,
    eigenVectors, elevatedIntermediateEigensystems,
    endTime, fmSolAssoc, freeBies,
    freeIenergiesAndMultiplets, fullHam, fullSolVec,
    ham, hamDim, hamEigenvaluesTemplate,
    hamString, indepSolVecVec, indepVars, intermediateHam,
    isolationValues, lin, linMat, ln, lnParams,
    logFilePrefix, magneticSimplifier,
    maxFreeEnergy, maxHistory, maxIterations,
    minFreeEnergy, minpoly,
    modelSymbols, multipletAssignments, needlePosition,
    numBlocks, solCompendium,
    openNotebooks, ordering, otherSimplifier, p0,
    paramBest, perHam,
    presentDataIndices, PrintFun, problemVarsPositions,
    problemVarsQ, problemVarsQString, problemVarsVec,
    problemVarsWithStartValues, reducedModelSymbols,
    roundedTruncationEnergy,
    runningInteractive, shiftToggle, simplifier,
    sol, solWithUncertainty,
    sortedTruncationIndex, sqdiff, standardValues,
    startTime,
    states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, timeTaken,
    truncatedIntermediateBasis, truncatedIntermediateHam,
    truncationEnergy, truncationIndices, RefParams,
    truncationUmbral, varHash,
    varsWithConstants, \[Lambda]0Vec,
    \[Lambda]exp
    },
  (
    \[Sigma]exp   = OptionValue["Energy Uncertainty in K"];
```

```
1099    solCompendium = <||>;
1100    refParamsVintage = OptionValue["RefParamsVintage"];
1101    RefParams = Which[
1102      refParamsVintage === "LaF3",
1103      LoadLaF3Parameters,
1104      refParamsVintage === "LiYF4",
1105      LoadLiYF4Parameters,
1106      True,
1107      refParamsVintage
1108    ];
1109    hamDim         = Binomial[14, numE];
1110    addShift       = OptionValue["AddConstantShift"];
1111    ln             = theLanthanides[[numE]];
1112    maxHistory     = OptionValue["MaxHistory"];
1113    maxIterations  = OptionValue["MaxIterations"];
1114    logFilePrefix  = If[OptionValue["FilePrefix"] == "",
1115                        ToString[theLanthanides[[numE]]],
1116                        OptionValue["FilePrefix"]
1117                      ];
1118    accuracyGoal  = OptionValue["AccuracyGoal"];
1119    PrintFun      = OptionValue["PrintFun"];
1120    freeIonSymbols       = OptionValue["FreeIonSymbols"];
1121    runningInteractive   = (Head[$ParentLink] === LinkObject);
1122    magneticSimplifier   = OptionValue["MagneticSimplifier"];
1123    magFieldSimplifier   = OptionValue["MagFieldSimplifier"];
1124    symmetrySimplifier   = OptionValue["SymmetrySimplifier"];
1125    otherSimplifier      = OptionValue["OtherSimplifier"];
1126    threeBodySimplifier  = If[Head[OptionValue["ThreeBodySimplifier"]]
       == Association,
1127                            OptionValue["ThreeBodySimplifier"][numE],
1128                            OptionValue["ThreeBodySimplifier"]
1129                          ];
1130
1131    truncationEnergy = If[OptionValue["TruncationEnergy"] ===
      Automatic,
1132      (
1133        PrintFun["Truncation energy set to Automatic, using the
      maximum energy (+20%) in the data ..."];
1134        Round[1.2 * Max[Select[First /@ expData, NumericQ[#] &]]]
1135      ),
1136      OptionValue["TruncationEnergy"]
1137      ];
1138    truncationEnergy = Max[50000, truncationEnergy];
1139    PrintFun["Using a truncation energy of ", truncationEnergy, " K"
      ];
1140
1141    simplifier = Join[magneticSimplifier,
1142                      magFieldSimplifier,
1143                      symmetrySimplifier,
1144                      threeBodySimplifier,
1145                      otherSimplifier];
1146
1147    PrintFun["Determining gaps in the data ..."];
1148    (* whatever is non-numeric is assumed as a known gap *)
1149    presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
      , ___}]];
1150    (* some indices omitted here based on the excludeDataIndices
      argument *)
1151    presentDataIndices = Complement[presentDataIndices,
      excludeDataIndices];
1152
1153    solCompendium["simplifier"]          = simplifier;
1154    solCompendium["excludeDataIndices"] = excludeDataIndices;
1155    solCompendium["startValues"]         = startValues;
1156    solCompendium["freeIonSymbols"]      = freeIonSymbols;
1157    solCompendium["truncationEnergy"]    = truncationEnergy;
1158    solCompendium["numE"]                = numE;
1159    solCompendium["expData"]             = expData;
1160    solCompendium["problemVars"]         = problemVars;
1161    solCompendium["maxIterations"]       = maxIterations;
1162    solCompendium["hamDim"]              = hamDim;
1163    solCompendium["constraints"]         = constraints;
1164
1165    modelSymbols  = Sort[Select[paramSymbols, Not[MemberQ[Join[
      racahSymbols, juddOfeltIntensitySymbols, chenSymbols,{t2Switch, \[
      Epsilon], gs, nE}],#]]&]];
```

```
1166    (* remove the symbols that will be removed by the simplifier, no
        symbol should remain here that is not in the symbolic Hamiltonian
        *)
1167    reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
        simplifier],#]]&];
1168
1169    (* this is useful to understand what are the arguments of the
        truncated compiled Hamiltonian *)
1170    If[OptionValue["SignatureCheck"],
1171      (
1172        PrintFun["Given the model parameters and the simplifying
        assumptions, the resultant model parameters are:"];
1173        PrintFun[{reducedModelSymbols}];
1174        PrintFun["Exiting ..."];
1175        Return[""];
1176      )
1177    ];
1178
1179    (* calculate the basis *)
1180    PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
1181    basis    = BasisLSJMJ[numE];
1182
1183    Which[refParamsVintage === Automatic,
1184      (
1185        PrintFun["Using the automatic vintage with freshly fitted
        free-ion parameters and others as in LaF3 ..."];
1186        lnParams      = LoadLaF3Parameters[ln];
1187        freeIonSol    = FreeIonSolver[expData, numE];
1188        freeIonParams = freeIonSol["bestParams"];
1189        lnParams      = Join[lnParams, freeIonParams];
1190      ),
1191      MemberQ[{List, Association}, Head[RefParams]],
1192      (
1193        RefParams = Association[RefParams];
1194        PrintFun["Using the given parameters as a starting point ..."
        ];
1195        lnParams    = RefParams;
1196        extraParams = LoadLaF3Parameters[ln];
1197        lnParams    = Join[extraParams, lnParams];
1198      ),
1199      True,
1200      (
1201        (* get the reference parameters from the given vintage *)
1202        PrintFun["Getting reference free-ion parameters for ", ln, "
        using ", refParamsVintage, " ..."];
1203        lnParams = ParamPad[RefParams[ln], "PrintFun" -> PrintFun];
1204      )
1205    ];
1206    freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
       numE];
1207    (* a more explicit alias *)
1208    allVars            = reducedModelSymbols;
1209    numericConstraints = Association@Select[constraints, NumericQ
        [#[[2]]] &];
1210    standardValues     = allVars /. Join[lnParams, numericConstraints
        ];
1211    solCompendium["allVars"]  = allVars;
1212    solCompendium["freeBies"] = freeBies;
1213
1214    (* reload compiled version if found *)
1215    varHash                   = Hash[{numE, allVars, freeBies,
        truncationEnergy, simplifier}];
1216    compiledIntermediateFname = ln <> "-compiled-intermediate-
        truncated-ham-" <> ToString[varHash] <> ".mx";
1217    compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
        compiledIntermediateFname}];
1218    solCompendium["compiledIntermediateFname"] =
        compiledIntermediateFname;
1219
1220    If[FileExistsQ[compiledIntermediateFname],
1221      PrintFun["This ion, free-ion params, and full set of variables
        have been used before (as determined by {numE, allVars, freeBies,
        truncationEnergy, simplifier}). Loading the previously saved
        compiled function and intermediate coupling basis ..."];
1222      PrintFun["Using : ", compiledIntermediateFname];
1223      {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
```

```
 Import[compiledIntermediateFname];,
(
  If[truncationEnergy == Infinity,
  (
      ham = EffectiveHamiltonian[numE, "ReturnInBlocks" -> False];
      theSimplifier = simplifier;
      ham = Normal@ReplaceInSparseArray[ham, simplifier];
      PrintFun["Compiling a function for the Hamiltonian with no truncation ..."];
      (* compile a function that will calculate the truncated Hamiltonian given the parameters in allVars, this is the function to be use in fitting *)
      compileIntermediateTruncatedHam = Compile[Evaluate[allVars], Evaluate[ham]];
      truncatedIntermediateBasis = SparseArray@IdentityMatrix[Binomial[14, numE]];
      (* save the compiled function *)
      PrintFun["Saving the compiled function for the Hamiltonian with no truncation and a placeholder intermediate basis ..."];
      Export[compiledIntermediateFname, {compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
  ),
  (
    (* grab the Hamiltonian preserving the block structure *)
    PrintFun["Assembling the Hamiltonian for f^",numE," keeping the block structure ..."];
    ham          = EffectiveHamiltonian[numE, "ReturnInBlocks"->True];
    (* apply the simplifier *)
    PrintFun["Simplifying using the aggregate set of simplification rules ..."];
    ham          = Map[ReplaceInSparseArray[#, simplifier]&, ham, {2}];
    PrintFun["Zeroing out every symbol in the Hamiltonian that is not a free-ion parameter ..."];
    (* Get the free ion symbols *)
    freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols, freeIonSymbols];
    (* Take the diagonal blocks for the intermediate analysis *)
    PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."];
    diagonalBlocks       = Diagonal[ham];
    (* simplify them to only keep the free ion symbols *)
    PrintFun["Simplifying the diagonal blocks to only keep the free ion symbols ..."];
    diagonalScalarBlocks = ReplaceInSparseArray[#, freeIonSimplifier]&/@diagonalBlocks;
    (* these include the MJ quantum numbers, remove that *)
    PrintFun["Contracting the basis vectors by removing the MJ quantum numbers from the diagonal blocks ..."];
    diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];

    argsOfTheIntermediateEigensystems         = StringJoin[Riffle[Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
    argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(ToString[#]<>"v") & /@ freeIonSymbols,", "]];
    PrintFun["argsOfTheIntermediateEigensystems = ", argsOfTheIntermediateEigensystems];
    PrintFun["argsForEvalInsideOfTheIntermediateSystems = ", argsForEvalInsideOfTheIntermediateSystems];
    PrintFun["(if the following fails, it might help to see if the arguments of TheIntermediateEigensystems match the ones shown above)"];

    (* compile a function that will effectively calculate the spectrum of all of the scalar blocks given the parameters of the free-ion part of the Hamiltonian *)
    (* compile one function for each of the blocks *)
    PrintFun["Compiling functions for the diagonal blocks of the Hamiltonian ..."];
    compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N[Normal[#]]]]&/@diagonalScalarBlocks;
    (* use that to create a function that will calculate the free-ion eigensystem *)
    TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζ
```

```
    v_] := (
1270        theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&) /@
    compiledDiagonal;
1271        theIntermediateEigensystems = Eigensystem /@
    theNumericBlocks;
1272        Js     = AllowedJ[numEv];
1273        basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
1274        (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
1275        elevatedIntermediateEigensystems = MapIndexed[EigenLever
    [#1, 2Js[[#2[[1]]]]+1 ]&, theIntermediateEigensystems];
1276        (* Identify a single MJ to keep *)
1277        pivot          = If[EvenQ[numEv],0,-1/2];
1278        LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
1279        (* calculate the multiplet assignments that the
    intermediate basis eigenvectors have *)
1280        needlePosition = 0;
1281        multipletAssignments = Table[
1282          (
1283            J         = Js[[idx]];
1284            eigenVecs = theIntermediateEigensystems[[idx]][[2]];
1285            majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
1286            majorComponentIndices       += needlePosition;
1287            needlePosition              += Length[
    majorComponentIndices];
1288            majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
1289            (* All of the degenerate eigenvectors belong to the
    same multiplet*)
1290            elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments, 2J+1];
1291            elevatedMultipletAssignments
1292          ),
1293        {idx, 1, Length[Js]}
1294        ];
1295        (* put together the multiplet assignments and the energies
    *)
1296        freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
1297        freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
1298        (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
1299        basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
1300        basisChanger = SparseArray[basisChanger];
1301        Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
1302      );
1303
1304      PrintFun["Calculating the intermediate eigensystems for ",ln,
    " using free-ion params from LaF3 ..."];
1305      (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
1306      {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
1307
1308      (* use that intermediate coupling basis to compile a function
     for the full Hamiltonian *)
1309      allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
1310      (* important that the intermediate coupling basis have
    attached energies, which make possible the truncation *)
1311      ordering = Ordering[allFreeEnergies];
1312      (* sort the free ion energies and determine which indices
    should be included in the truncation *)
1313      allFreeEnergiesSorted          = Sort[allFreeEnergies];
1314      {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
1315      (* determine the index at which the energy is equal or larger
     than the truncation energy *)
1316      sortedTruncationIndex = Which[
1317        truncationEnergy > (maxFreeEnergy-minFreeEnergy),
```

```
1318            hamDim,
1319            True,
1320            FirstPosition[allFreeEnergiesSorted - Min[
    allFreeEnergiesSorted],x_/;x>truncationEnergy,{0},1][[1]]
1321        ];
1322        (* the actual energy at which the truncation is made *)
1323        roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
1324
1325        (* the indices that participate in the truncation *)
1326        truncationIndices = ordering[[;;sortedTruncationIndex]];
1327        PrintFun["Computing the block structure of the change of
    basis array ..."];
1328        blockSizes          = BlockArrayDimensionsArray[ham];
1329        basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
1330        blockShifts         = First /@ Diagonal[blockSizes];
1331        numBlocks           = Length[blockSizes];
1332        (* using the ham (with all the symbols) change the basis to
    the computed one *)
1333        PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
1334        intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks
    ];
1335        PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
1336        Do[
1337          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1338          {rowIdx, 1, numBlocks},
1339          {colIdx, 1, numBlocks}
1340        ];
1341        intermediateHam = BlockMatrixMultiply[BlockTranspose[
    basisChangerBlocks],intermediateHam];
1342        PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
1343        Do[
1344          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1345          {rowIdx, 1, numBlocks},
1346          {colIdx, 1, numBlocks}
1347        ];
1348        (* using the truncation indices truncate that one *)
1349        PrintFun["Truncating the Hamiltonian ..."];
1350        truncatedIntermediateHam = TruncateBlockArray[intermediateHam
    , truncationIndices, blockShifts];
1351        (* these are the basis vectors for the truncated hamiltonian
    *)
1352        PrintFun["Saving the truncated intermediate basis ..."];
1353        truncatedIntermediateBasis = basisChanger[[All,
    truncationIndices]];
1354
1355        PrintFun["Compiling a function for the truncated Hamiltonian
    ..."];
1356        (* compile a function that will calculate the truncated
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
1357        compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
    Evaluate[truncatedIntermediateHam]];
1358        (* save the compiled function *)
1359        PrintFun["Saving the compiled function for the truncated
    Hamiltonian and the truncated intermediate basis ..."];
1360        Export[compiledIntermediateFname, {
    compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
1361      )
1362    ];
1363    )
1364    ];
1365
1366    truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
1367    PrintFun["The truncated Hamiltonian has a dimension of ",
    truncationUmbral, "x", truncationUmbral, " ..."];
1368    presentDataIndices = Select[presentDataIndices, # <=
    truncationUmbral &];
1369    solCompendium["presentDataIndices"] = presentDataIndices;
1370
1371    (* the problemVars are the symbols that will be fitted for *)
```

```
1372
1373    PrintFun["Starting up the fitting process using the Levenberg-
        Marquardt method ..."];
1374    (* using the problemVars I need to create the argument list
        including _?NumericQ *)
1375    problemVarsQ       = (ToString[#] <> "_?NumericQ") & /@
        problemVars;
1376    problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
1377    (* we also need to have the padded arguments with the variables
        in the right position and the fixed values in the remaining ones
        *)
1378    problemVarsPositions = Position[allVars, #][[1, 1]] & /@
        problemVars;
1379    problemVarsString    = StringJoin[Riffle[ToString /@ problemVars,
         ", "]];
1380    (* to feed parameters to the Hamiltonian, which includes all
        parameters, we need to form the set of arguments, with fixed
        values where needed, and the variables in the right position *)
1381    varsWithConstants                          = standardValues;
1382    varsWithConstants[[problemVarsPositions]] = problemVars;
1383    varsWithConstantsString                    = ToString[
        varsWithConstants];
1384
1385    (* this following function serves eigenvalues from the
        Hamiltonian, has memoization so it might grow to use a lot of RAM
        *)
1386    Clear[HamSortedEigenvalues];
1387    hamEigenvaluesTemplate = StringTemplate["
1388    HamSortedEigenvalues[`problemVarsQ`]:=(
1389      ham          = compileIntermediateTruncatedHam@@`
        varsWithConstants`;
1390      eigenValues = Chop[Sort@Eigenvalues@ham];
1391      eigenValues = eigenValues - Min[eigenValues];
1392      HamSortedEigenvalues[`problemVarsString`] = eigenValues;
1393      Return[eigenValues]
1394    )"];
1395    hamString = hamEigenvaluesTemplate[<|
1396        "problemVarsQ"      -> problemVarsQString,
1397        "varsWithConstants" -> varsWithConstantsString,
1398        "problemVarsString" -> problemVarsString
1399        |>];
1400    ToExpression[hamString];
1401
1402    (* we also need a function that will pick the i-th eigenvalue,
        this seems unnecessary but it's needed to form the right
        functional form expected by the Levenberg-Marquardt method *)
1403    eigenvalueDispenserTemplate = StringTemplate["
1404    PartialHamEigenvalues[`problemVarsQ`][i_]:=(
1405      eigenVals = HamSortedEigenvalues[`problemVarsString`];
1406      eigenVals[[i]]
1407    )
1408    "];
1409    eigenValueDispenserString = eigenvalueDispenserTemplate[<|
1410        "problemVarsQ"      -> problemVarsQString,
1411        "problemVarsString" -> problemVarsString
1412        |>];
1413    ToExpression[eigenValueDispenserString];
1414
1415    PrintFun["Determining the free variables after constraints ..."];
1416    constrainedProblemVars     = (problemVars /. constraints);
1417    constrainedProblemVarsList = Variables[constrainedProblemVars];
1418    If[addShift,
1419      PrintFun["Adding a constant shift to the fitting parameters ...
        "];
1420      constrainedProblemVarsList = Append[constrainedProblemVarsList,
         \[Epsilon]]
1421    ];
1422
1423    indepVars = Complement[problemVars, #[[1]] & /@ constraints];
1424    stringPartialVars = ToString/@constrainedProblemVarsList;
1425
1426    paramSols  = {};
1427    rmsHistory = {};
1428    steps      = 0;
1429    problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
1430    If[addShift,
```

```
    problemVarsWithStartValues = Append[problemVarsWithStartValues, {\[Epsilon] ,0}];
  ];
  openNotebooks     = If[runningInteractive,
            ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks[],
            {}];
  If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["ProgressView"],
    ProgressNotebook["Basic"->False]
  ];
  degressOfFreedom = Length[presentDataIndices] - Length[problemVars] - 1;
  PrintFun["Fitting for ", Length[presentDataIndices], " data points with ", Length[problemVars], " free parameters.", " The effective degrees of freedom are ", degressOfFreedom, " ..."];

  PrintFun["Fitting model to data ..."];
  startTime   = Now;
  shiftToggle = If[addShift, 1, 0];
  sol = FindMinimum[
    Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
    {j, presentDataIndices}],
    problemVarsWithStartValues,
    Method         -> "LevenbergMarquardt",
    MaxIterations  -> OptionValue["MaxIterations"],
    AccuracyGoal   -> OptionValue["AccuracyGoal"],
    StepMonitor :> (
      steps      += 1;
      currentSqSum = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j, presentDataIndices}];
      currentRMS = Sqrt[currentSqSum / degressOfFreedom];
      paramSols  = AddToList[paramSols, constrainedProblemVarsList, maxHistory];
      rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
      )
    ];
  endTime = Now;
  timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
  PrintFun["Solution found in ", timeTaken, "s"];

  solVec = constrainedProblemVars /. sol[[-1]];
  indepSolVec = indepVars /. sol[[-1]];
  If[addShift,
    \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
    \[Epsilon]Best = 0
  ];
  fullSolVec = standardValues;
  fullSolVec[[problemVarsPositions]] = solVec;
  PrintFun["Calculating the truncated numerical Hamiltonian corresponding to the solution ..."];
  fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
  PrintFun["Calculating energies and eigenvectors ..."];
  {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
  states = Transpose[{eigenEnergies, eigenVectors}];
  states = SortBy[states, First];
  eigenEnergies = First /@ states;
  PrintFun["Shifting energies to make ground state zero of energy ..."];
  eigenEnergies = eigenEnergies - eigenEnergies[[1]];
  PrintFun["Calculating the linear approximant to each eigenvalue ..."];
  allVarsVec = Transpose[{allVars}];
  p0 = Transpose[{fullSolVec}];
  linMat = {};
  If[addShift,
    tail = -2,
    tail = -1];
  Do[
    (
      aVarPosition = Position[allVars, aVar][[1, 1]];
      isolationValues = ConstantArray[0, Length[allVars]];
      isolationValues[[aVarPosition]] = 1;
      dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
```

```
    constraints]];
1494        Do[
1495          isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
1496          {dVar, dependentVars}
1497        ];
1498        perHam  = compileIntermediateTruncatedHam @@ isolationValues;
1499        lin     = FirstOrderPerturbation[Last /@ states, perHam];
1500        linMat  = Append[linMat, lin];
1501      ),
1502      {aVar, constrainedProblemVarsList[[;;tail]]}
1503    ];
1504    PrintFun["Removing the gradient of the ground state ..."];
1505    linMat = (# - #[[1]] & /@ linMat);
1506    PrintFun["Transposing derivative matrices into columns ..."];
1507    linMat  = Transpose[linMat];
1508
1509    PrintFun["Calculating the eigenvalue vector at solution ..."];
1510    \[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
1511    PrintFun["Putting together the experimental vector ..."];
1512    \[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
    ]]}];
1513    problemVarsVec = If[addShift,
1514      Transpose[{constrainedProblemVarsList[[;;-2]]}],
1515      Transpose[{constrainedProblemVarsList}]
1516    ];
1517    indepSolVecVec = Transpose[{indepSolVec}];
1518    PrintFun["Calculating the difference between eigenvalues at
    solution ..."];
1519    diff = If[linMat == {},
1520      (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
1521      (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
    presentDataIndices]].(problemVarsVec - indepSolVecVec)
1522    ];
1523    PrintFun["Calculating the sum of squares of differences around
    solution ... "];
1524    sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
1525    PrintFun["Calculating the minimum (which should coincide with sol
    ) ..."];
1526    minpoly       = sqdiff /. AssociationThread[problemVars -> solVec
    ];
1527    fmSolAssoc    = Association[sol[[2]]];
1528    If[\[Sigma]exp == Automatic,
1529      \[Sigma]exp = Sqrt[minpoly / degressOfFreedom];
1530    ];
1531    \[CapitalDelta]\[Chi]2 = Sqrt[degressOfFreedom];
1532    Amat = (1/\[Sigma]exp^2) * Transpose[linMat[[presentDataIndices
    ]]].linMat[[presentDataIndices]];
1533    paramIntervals = EllipsoidBoundingBox[Amat, \[CapitalDelta]\[Chi
    ]2];
1534    PrintFun["Calculating the uncertainty in the parameters ..."];
1535    solWithUncertainty = Table[
1536      (
1537        aVar        = constrainedProblemVarsList[[varIdx]];
1538        paramBest   = aVar /. fmSolAssoc;
1539        (aVar -> {paramBest, paramIntervals[[varIdx, 2]]})
1540      ),
1541    {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
1542    ];
1543
1544    bestRMS    = Sqrt[minpoly / degressOfFreedom];
1545    bestParams = sol[[2]];
1546    bestWithConstraints = Association@Join[constraints, bestParams];
1547    bestWithConstraints = bestWithConstraints /. bestWithConstraints;
1548    bestWithConstraints = (# + 0.) & /@ bestWithConstraints;
1549
1550    solCompendium["degreesOfFreedom"]    = degressOfFreedom;
1551    solCompendium["solWithUncertainty"]  = solWithUncertainty;
1552    solCompendium["truncatedDim"]        = truncationUmbral;
1553    solCompendium["fittedLevels"]        = Length[presentDataIndices
    ];
1554    solCompendium["actualSteps"]         = steps;
1555    solCompendium["bestRMS"]             = bestRMS;
1556    solCompendium["problemVars"]         = problemVars;
1557    solCompendium["paramSols"]           = paramSols;
1558    solCompendium["rmsHistory"]          = rmsHistory;
```

```
    solCompendium["Appendix"]             = OptionValue["
    AppendToLogFile"];
    solCompendium["timeTaken/s"]          = timeTaken;
    solCompendium["bestParams"]           = bestParams;
    solCompendium["bestParamsWithConstraints"] = bestWithConstraints;

    If[OptionValue["SaveEigenvectors"],
        solCompendium["states"] = {#[[1]] + \[Epsilon]Best, #[[2]]}
    &/@ (Chop /@ ShiftedLevels[states]),
        (
          finalEnergies = Sort[First /@ states];
          finalEnergies = finalEnergies - finalEnergies[[1]];
          finalEnergies = finalEnergies + \[Epsilon]Best;
          finalEnergies  = Chop /@ finalEnergies;
          solCompendium["energies"] = finalEnergies;
      )
    ];
    If[OptionValue["SaveToLog"],
      PrintFun["Saving the solution to the log file ..."];
      LogSol[solCompendium, logFilePrefix];
    ];
    PrintFun["Finished ..."];
    Return[solCompendium];
  )
];


MostlyOrthogonalFit::usage="MostlyOrthogonalFit[numE, expData,
    excludeDataIndices, problemVars, startValues, \[Sigma]exp,
    constraints_List, Options] fits the given expData in an f^numE
    configuration, by using the symbols in problemVars. The symbols
    given in problemVars may be constrained or held constant, this
    being controlled by constraints list which is a list of
    replacement rules expressing desired constraints. The constraints
    list additional constraints imposed upon the model parameters that
     remain once other simplifications have been \"baked\" into the
    compiled Hamiltonians that are used to increase the speed of the
    calculation.

Important, note that in the case of odd number of electrons the given
     data must explicitly include the Kramers degeneracy;
    excludeDataIndices must be compatible with this.

The list expData needs to be a list of lists with the only
    restriction that the first element of them corresponds to energies
     of levels. In this list, an empty value can be used to indicate
    known gaps in the data. Even if the energy value for a level is
    known (and given in expData) certain values can be omitted from
    the fitting procedure through the list excludeDataIndices, which
    correspond to indices in expData that should be skipped over.

The Hamiltonian used for fitting is version that has been truncated
    either by using the maximum energy given in expData or by manually
     setting a truncation energy using the option \"TruncationEnergy\"
    .

The argument \[Sigma]exp is the estimated uncertainty in the
    differences between the calculated and the experimental energy
    levels. This is used to estimate the uncertainty in the fitted
    parameters. Admittedly this will be a rough estimate (at least on
    the contribution of the calculated uncertainty), but it is better
    than nothing and may at least provide a lower bound to the
    uncertainty in the fitted parameters. It is assumed that the
    uncertainty in the differences between the calculated and the
    experimental energy levels is the same for all of them.

The list startValues is a list with all of the parameters needed to
    define the Hamiltonian (including the initial values for
    problemVars).

The function saves the solution to a file. The file is named with a
    prefix (controlled by the option \"FilePrefix\") and a UUID. The
    file is saved in the log sub-directory as a .m file.

Here's a description of the different parts of this function: first
    the Hamiltonian is assembled and simplified using the given
```

```
    simplifications. Then the intermediate coupling basis is
    calculated using the free-ion parameters for the given lanthanide.
     The Hamiltonian is then changed to the intermediate coupling
    basis and truncated. The truncated Hamiltonian is then compiled
    into a function that can be used to calculate the energy levels of
     the truncated Hamiltonian. The function that calculates the
    energy levels is then used to fit the experimental data. The
    fitting is done using FindMinimum with the Levenberg-Marquardt
    method.
1599
1600 The function returns an association with the following keys:
1601
1602 - \"bestRMS\" which is the best \[Sigma] value found.
1603 - \"bestParams\" which is the best set of parameters found for the
    variables that were not constrained.
1604 - \"bestParamsWithConstraints\" which has the best set of parameters
    (from - \"bestParams\") together with the used constraints. These
    include all the parameters in the model, even those that were not
    fitted for.
1605 - \"paramSols\" which is a list of the parameters trajectories during
     the stepping of the fitting algorithm.
1606 - \"timeTaken/s\" which is the time taken to find the best fit.
1607 - \"simplifier\" which is the simplifier used to simplify the
    Hamiltonian.
1608 - \"excludeDataIndices\" as given in the input.
1609 - \"startValues\" as given in the input.
1610
1611 - \"freeIonSymbols\" which are the symbols used in the intermediate
    coupling basis.
1612 - \"truncationEnergy\" which is the energy used to truncate the
    Hamiltonian, if it was set to Automatic, the value here is the
    actual energy used.
1613 - \"numE\" which is the number of electrons in the f^numE
    configuration.
1614 - \"expData\" which is the experimental data used for fitting.
1615 - \"problemVars\" which are the symbols considered for fitting
1616
1617 - \"maxIterations\" which is the maximum number of iterations used by
     NMinimize.
1618 - \"hamDim\" which is the dimension of the full Hamiltonian.
1619 - \"allVars\" which are all the symbols defining the Hamiltonian
    under the aggregate simplifications.
1620 - \"freeBies\" which are the free-ion parameters used to define the
    intermediate coupling basis.
1621 - \"truncatedDim\" which is the dimension of the truncated
    Hamiltonian.
1622 - \"compiledIntermediateFname\" the file name of the compiled
    function used for the truncated Hamiltonian.
1623
1624 - \"fittedLevels\" which is the number of levels fitted for.
1625 - \"actualSteps\" the number of steps that FindMiniminum actually
    took.
1626 - \"solWithUncertainty\" which is a list of replacement rules of the
    form (paramSymbol -> {bestEstimate, uncertainty}).
1627 - \"rmsHistory\" which is a list of the \[Sigma] values found during
    the fitting.
1628 - \"Appendix\" which is an association appended to the log file under
     the key \"Appendix\".
1629 - \"presentDataIndices\" which is the list of indices in expData that
     were used for fitting, this takes into account both the empty
    indices in expData and also the indices in excludeDataIndices.
1630
1631 - \"states\" which contains a list of eigenvalues and eigenvectors
    for the fitted model, this is only available if the option \"
    SaveEigenvectors\" is set to True; if a general shift of energy
    was allowed for in the fitting, then the energies are shifted
    accordingly.
1632 - \"energies\" which is a list of the energies of the fitted levels,
    this is only available if the option \"SaveEigenvectors\" is set
    to False. If a general shift of energy was allowed for in the
    fitting, then the energies are shifted accordingly.
1633 - \"degreesOfFreedom\" which is equal to the nunber of fitted state
    energies minus the number of parameters used in fitting.
1634
1635 The function admits the following options with corresponding default
    values:
```

```
- \"MaxHistory\" : determines how long the logs for the solver can be.
- \"MaxIterations\": determines the maximum number of iterations used by NMinimize.
- \"FilePrefix\" : the prefix to use for the subfolder in the log filder, in which the solution files are saved, by default this is \"calcs\" so that the calculation files are saved under the directory \"log/calcs\".
- \"AddConstantShift\" : if True then a constant shift is allowed in the fitting, default is False. If this is the case the variable \"\[Epsilon]\" is added to the list of variables to be fitted for, it must not be included in problemVars.

- \"AccuracyGoal\": the accuracy goal used by NMinimize, default of 5.
- \"TrucationEnergy\": if Automatic then the maximum energy in expData is taken, else it takes the value set by this option. In all cases the energies in expData are only considered up to this value.
- \"PrintFun\": the function used to print progress messages, the default is PrintTemporary.
- \"RefParamsVintage\": the vintage of the reference parameters to use. The reference parameters are both used to determine the truncated Hamiltonian, and also as starting values for the solver. It may be \"LaF3\", in which case reference parameters from Carnall are used. It may also be \"LiYF4\", in which case the reference parameters from the LiYF4 paper are used. It may also be Automatic, in which case the given experimental data is used to determine starting values for F^k and ζ. It may also be a list or association that provides values for the Slater integrals and spin-orbit coupling, the remaining necessary parameters complemented by using \"LaF3\".

- \"ProgressView\": whether or not a progress window will be opened to show the progress of the solver, the default is True.
- \"SignatureCheck\": if True then then the function returns prematurely, returning a list with the symbols that would have defined the Hamiltonian after all simplifications have been applied. Useful to check the entire parameter set that the Hamiltonian has.
- \"SaveEigenvectors\": if True then the both the eigenvectors and eigenvalues are saved under the \"states\" key of the returned association. If False then only the energies are saved, the default is False.

- \"AppendToLogFile\": an association appended to the log file under the key \"Appendix\".
- \"MagneticSimplifier\": a list of replacement rules to simplify the Marvin and pesudo-magnetic paramters. Here the ratios of the Marvin parameters and the pseudo-magnetic parameters are defined to simplify the magnetic part of the Hamiltonian.
- \"MagFieldSimplifier\": a list of replacement rules to specify a magnetic field (in T), if set to {}, then {Bx, By, Bz} can also be used as variables to be fitted for.

- \"SymmetrySimplifier\": a list of replacements rules to simplify the crystal field.
- \"OtherSimplifier\": an additional list of replacement rules that are applied to the Hamiltonian before computing with it. Here the spin-spin contribution can be turned off by setting \[Sigma]SS->0, which is the default.
";
Options[MostlyOrthogonalFit] = {
  "MaxHistory"       -> 200,
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "calcs",
  "ProgressView"     -> True,
  "TruncationEnergy" -> Automatic,
  "AccuracyGoal"     -> 5,
  "PrintFun"         -> PrintTemporary,
  "RefParamsVintage" -> "LaF3",
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "SaveToLog"        -> False,
```

```
  "Energy Uncertainty in K" -> Automatic,
  "MagneticSimplifier" -> {
    M2 -> 56/100 M0,
    M4 -> 31/100 M0,
    P4 -> 1/2 P2,
    P6 -> 1/10 P2
    },
  "MagFieldSimplifier" -> {
    Bx -> 0,
    By -> 0,
    Bz -> 0
    },
  "SymmetrySimplifier" -> {
    B12->0, B14->0, B16->0, B34->0, B36->0, B56->0,
    S12->0 ,S14->0, S16->0, S22->0, S24->0, S26->0,
    S34->0 ,S36->0, S44->0, S46->0, S56->0, S66->0
    },
  "OtherSimplifier" -> {
    E0p->0,
    P0->0,
    \[Sigma]SS->0,
    T11p->0,
    T12->0,
    T14->0,
    T15->0,
    T16->0,
    T18->0,
    T17->0,
    T19->0,
    wChErrA ->0,
    wChErrB->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0,
      T3->0,
      T4->0,
      T6->0,
      T7->0,
      T8->0,
      T11p->0,
      T12->0,
      T14->0,
      T15->0,
      T16->0,
      T18->0,
      T17->0,
      T19->0,
      T2p->0},
    2 -> {
      T2->0,
      T3->0,
      T4->0,
      T6->0,
      T7->0,
      T8->0,
      T11p->0,
      T12->0,
      T14->0,
      T15->0,
      T16->0,
      T18->0,
      T17->0,
      T19->0,
      T2p->0
      },
    3  -> {t2Switch -> 1},
    4  -> {t2Switch -> 1},
    5  -> {t2Switch -> 1},
    6  -> {t2Switch -> 1},
    7  -> {t2Switch -> 1},
    8  -> {t2Switch -> 0},
    9  -> {t2Switch -> 0},
    10 -> {t2Switch -> 0},
    11 -> {t2Switch -> 0},
    12 -> {
```

```
      t2Switch -> 0,
      T2->0,
      T2p->0,
      T3->0,
      T4->0,
      T6->0,
      T7->0,
      T8->0,
      T11p->0,
      T12->0,
      T14->0,
      T15->0,
      T16->0,
      T18->0,
      T17->0,
      T19->0
      },
    13->{
      t2Switch -> 0,
      T2->0,
      T2p->0,
      T3->0,
      T4->0,
      T6->0,
      T7->0,
      T8->0,
      T11p->0,
      T12->0,
      T14->0,
      T15->0,
      T16->0,
      T18->0,
      T17->0,
      T19->0
      }
    |>,
    "FreeIonSymbols" -> {E0p, E1p, E2p, E3p, ζ}
};
MostlyOrthogonalFit[numE_Integer, expData_List,
    excludeDataIndices_List, problemVars_List, startValues_Association
    , constraints_List, OptionsPattern[]]:=Module[
  {accuracyGoal,
  allFreeEnergies,
  allFreeEnergiesSorted,
  allVars,
  allVarsVec,
  argsForEvalInsideOfTheIntermediateSystems,
  argsOfTheIntermediateEigensystems,
  aVar,
  aVarPosition,
  basis,

  basisChanger,
  basisChangerBlocks,
  bestParams,
  bestRMS,
  blockShifts,
  blockSizes,
  compiledDiagonal,
  compiledIntermediateFname,
  constrainedProblemVars,
  constrainedProblemVarsList,

  currentRMS,
  degressOfFreedom,
  dependentVars,
  diagonalBlocks,
  diagonalScalarBlocks,
  diff,
  eigenEnergies,
  eigenvalueDispenserTemplate,
  eigenVectors,
  elevatedIntermediateEigensystems,

  endTime,
  fmSolAssoc,
```

```
  freeBies,
  freeIenergiesAndMultiplets,
  fullHam,
  fullSolVec,
  ham,
  hamDim,
  hamEigenvaluesTemplate,
  hamString,

  indepSolVecVec,
  indepVars,
  intermediateHam,
  isolationValues,
  lin,
  linMat,
  ln,
  lnParams,
  logFilePrefix,
  magneticSimplifier,

  maxFreeEnergy,
  maxHistory,
  maxIterations,
  minFreeEnergy,
  minpoly,
  modelSymbols,
  multipletAssignments,
  needlePosition,
  numBlocks,
  openNotebooks,

  ordering,
  otherSimplifier,
  p0,
  paramBest,
  perHam,
  presentDataIndices,
  PrintFun,
  problemVarsPositions,
  problemVarsQ,
  problemVarsQString,

  problemVarsVec,
  problemVarsWithStartValues,
  reducedModelSymbols,
  RefParams,
  roundedTruncationEnergy,
  runningInteractive,
  shiftToggle,
  simplifier,
  sol,
  solCompendium,

  solWithUncertainty,
  sortedTruncationIndex,
  sqdiff,
  standardValues,
  startTime,
  states,
  steps,
  symmetrySimplifier,
  theIntermediateEigensystems,
  TheIntermediateEigensystems,

  timeTaken,
  truncatedIntermediateBasis,
  truncatedIntermediateHam,
  truncationEnergy,
  truncationIndices,
  truncationUmbral,
  varHash,
  varsWithConstants,
  \[Lambda]0Vec,
  \[Lambda]exp
},
  (
```

```
    \[Sigma]exp   = OptionValue["Energy Uncertainty in K"];
    refParamsVintage = OptionValue["RefParamsVintage"];
    RefParams = Which[
      refParamsVintage === "LaF3",
      LoadLaF3Parameters,
      refParamsVintage === "LiYF4",
      LoadLiYF4Parameters,
      True,
      refParamsVintage
    ];
    solCompendium = <||>;
    hamDim        = Binomial[14, numE];
    addShift      = OptionValue["AddConstantShift"];
    ln            = theLanthanides[[numE]];
    maxHistory    = OptionValue["MaxHistory"];
    maxIterations = OptionValue["MaxIterations"];
    logFilePrefix = If[OptionValue["FilePrefix"] == "",
                       ToString[theLanthanides[[numE]]],
                       OptionValue["FilePrefix"]
                     ];
    accuracyGoal = OptionValue["AccuracyGoal"];
    PrintFun     = OptionValue["PrintFun"];
    freeIonSymbols      = OptionValue["FreeIonSymbols"];
    runningInteractive  = (Head[$ParentLink] === LinkObject);
    magneticSimplifier  = OptionValue["MagneticSimplifier"];
    magFieldSimplifier  = OptionValue["MagFieldSimplifier"];
    symmetrySimplifier  = OptionValue["SymmetrySimplifier"];
    otherSimplifier     = OptionValue["OtherSimplifier"];
    threeBodySimplifier = If[Head[OptionValue["ThreeBodySimplifier"]]
     == Association,
                             OptionValue["ThreeBodySimplifier"][numE],
                             OptionValue["ThreeBodySimplifier"]
                           ];

    truncationEnergy = If[OptionValue["TruncationEnergy"] ===
    Automatic,
      (
        PrintFun["Truncation energy set to Automatic, using the
    maximum energy (+20%) in the data ..."];
        Round[1.2 * Max[Select[First /@ expData, NumericQ[#] &]]]
      ),
      OptionValue["TruncationEnergy"]
      ];
    truncationEnergy = Max[50000, truncationEnergy];
    PrintFun["Using a truncation energy of ", truncationEnergy, " K"
    ];

    simplifier = Join[magneticSimplifier,
                      magFieldSimplifier,
                      symmetrySimplifier,
                      threeBodySimplifier,
                      otherSimplifier];

    PrintFun["Determining gaps in the data ..."];
    (* whatever is non-numeric is assumed as a known gap *)
    presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
    , ___}]];
    (* some indices omitted here based on the excludeDataIndices
    argument, note that presentDataIndices is somewhat a misnomer*)
    presentDataIndices = Complement[presentDataIndices,
    excludeDataIndices];

    solCompendium["simplifier"]         = simplifier;
    solCompendium["excludeDataIndices"] = excludeDataIndices;
    solCompendium["startValues"]        = startValues;
    solCompendium["freeIonSymbols"]     = freeIonSymbols;
    solCompendium["truncationEnergy"]   = truncationEnergy;
    solCompendium["numE"]               = numE;
    solCompendium["expData"]            = expData;
    solCompendium["problemVars"]        = problemVars;
    solCompendium["maxIterations"]      = maxIterations;
    solCompendium["hamDim"]             = hamDim;
    solCompendium["constraints"]        = constraints;

    modelSymbols  = Select[paramSymbols,
      Not[MemberQ[Join[racahSymbols,
```

```
    juddOfeltIntensitySymbols,
    slaterSymbols,
    {α, β, γ},
    {T2},
    chenSymbols,
    {t2Switch, \[Epsilon], gs, nE}],#]]&
  ];
  modelSymbols = Sort[modelSymbols];
  (* remove the symbols that will be removed by the simplifier, no
  symbol should remain here that is not in the symbolic Hamiltonian
  *)
  reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
  simplifier],#]]&];

  (* this is useful to understand what are the arguments of the
  truncated compiled Hamiltonian *)
  If[OptionValue["SignatureCheck"],
    (
      PrintFun["Given the model parameters and the simplifying
  assumptions, the resultant model parameters are:"];
      PrintFun[{reducedModelSymbols}];
      PrintFun["Exiting ..."];
      Return[""];
    )
  ];

  (* calculate the basis *)
  PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
  basis    = BasisLSJMJ[numE];

  Which[refParamsVintage === Automatic,
    (
      PrintFun["This is not currently supported, please provide a
  vintage for the reference parameters."];
      Return[$Failed];
    ),
    MemberQ[{List, Association}, Head[RefParams]],
    (
      RefParams = Association[RefParams];
      PrintFun["Using the given parameters as a starting point ..."
  ];
      lnParams    = RefParams;
      extraParams = FromNonOrthogonalToMostlyOrthogonal[
  LoadLaF3Parameters[ln], numE];
      lnParams    = Join[extraParams, lnParams];
    ),
    True,
    (
      (* get the reference parameters from the given vintage *)
      PrintFun["Getting reference free-ion parameters for ", ln, "
  using ", refParamsVintage, " ..."];
      lnParams = ParamPad[FromNonOrthogonalToMostlyOrthogonal[
  RefParams[ln], numE],
        "PrintFun" -> PrintFun];
    )
  ];

  freeBies = Prepend[Values[(# -> (# /. lnParams)) &/@
  freeIonSymbols],  numE];

  (* a more explicit alias *)
  allVars            = reducedModelSymbols;
  numericConstraints = Association@Select[constraints, NumericQ
  [#[[2]]] &];
  standardValues     = allVars /. Join[lnParams, numericConstraints
  ];
  solCompendium["allVars"]  = allVars;
  solCompendium["freeBies"] = freeBies;

  (* reload compiled version if found *)
  varHash                   = Hash[{numE, allVars, freeBies,
  truncationEnergy, simplifier}];
  compiledIntermediateFname = ln <> "-compiled-intermediate-
  truncated-ham-" <> ToString[varHash] <> ".mx";
  compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
  compiledIntermediateFname}];
```

```
2026    solCompendium["compiledIntermediateFname"] =
        compiledIntermediateFname;
2027
2028    If[FileExistsQ[compiledIntermediateFname],
2029    (
2030      PrintFun["This ion, free-ion params, and full set of variables
        have been used before (as determined by {numE, allVars, freeBies,
        truncationEnergy, simplifier}). Loading the previously saved
        compiled function and intermediate coupling basis ..."];
2031      PrintFun["Using : ", compiledIntermediateFname];
2032      {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
         Import[compiledIntermediateFname];
2033    ),
2034    (
2035      If[truncationEnergy == Infinity,
2036      (
2037          ham = EffectiveHamiltonian[numE,
2038                "ReturnInBlocks" -> False,
2039                "OperatorBasis" -> "MostlyOrthogonal"];
2040          theSimplifier = simplifier;
2041          ham = Normal @ ReplaceInSparseArray[ham, simplifier];
2042          PrintFun["Compiling a function for the Hamiltonian with no
        truncation ..."];
2043          (* compile a function that will calculate the truncated
        Hamiltonian given the parameters in allVars, this is the function
        to be use in fitting *)
2044          compileIntermediateTruncatedHam = Compile[Evaluate[allVars
        ], Evaluate[ham]];
2045          truncatedIntermediateBasis      =
        SparseArray@IdentityMatrix[Binomial[14, numE]];
2046          (* save the compiled function *)
2047          PrintFun["Saving the compiled function for the Hamiltonian
        with no truncation and a placeholder intermediate basis ..."];
2048          Export[compiledIntermediateFname, {
        compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
2049      ),
2050      (
2051        (* grab the Hamiltonian preserving the block structure *)
2052        PrintFun["Assembling the Hamiltonian for f^", numE, " keeping
         the block structure ..."];
2053        ham           = EffectiveHamiltonian[numE,
2054                          "ReturnInBlocks"->True,
2055                          "OperatorBasis"->"MostlyOrthogonal"];
2056        (* apply the simplifier *)
2057        PrintFun["Simplifying using the aggregate set of
        simplification rules ..."];
2058        ham           = Map[ReplaceInSparseArray[#, simplifier]&, ham,
         {2}];
2059        PrintFun["Zeroing out every symbol in the Hamiltonian that is
         not a free-ion parameter ..."];
2060
2061        (* Get the free ion symbols *)
2062        freeIonSimplifier = (#->0) & /@ Complement[
        reducedModelSymbols, freeIonSymbols];
2063
2064        (* Take the diagonal blocks for the intermediate analysis *)
2065        PrintFun["Grabbing the diagonal blocks of the Hamiltonian ...
        "];
2066        diagonalBlocks       = Diagonal[ham];
2067
2068        (* simplify them to only keep the free ion symbols *)
2069        PrintFun["Simplifying the diagonal blocks to only keep the
        free ion symbols ..."];
2070        diagonalScalarBlocks = ReplaceInSparseArray[#,
        freeIonSimplifier] &/@ diagonalBlocks;
2071
2072        (* these include the MJ quantum numbers, remove that *)
2073        PrintFun["Contracting the basis vectors by removing the MJ
        quantum numbers from the diagonal blocks ..."];
2074        diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
2075
2076        argsOfTheIntermediateEigensystems         = StringJoin[Riffle
        [Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
2077        argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle
        [(ToString[#]<>"v") & /@ freeIonSymbols,", "]];
2078        PrintFun["argsOfTheIntermediateEigensystems = ",
```

```
    argsOfTheIntermediateEigensystems];
2079        PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
    argsForEvalInsideOfTheIntermediateSystems];
2080        PrintFun["(if the following fails, it might help to see if
    the arguments of TheIntermediateEigensystems match the ones shown
    above)"];
2081
2082        (* compile a function that will effectively calculate the
    spectrum of all of the scalar blocks given the parameters of the
    free-ion part of the Hamiltonian *)
2083        (* compile one function for each of the blocks *)
2084        PrintFun["Compiling functions for the diagonal blocks of the
    Hamiltonian ..."];
2085        compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate
    [N[Normal[#]]]]& /@ diagonalScalarBlocks;
2086        (* use that to create a function that will calculate the free
    -ion eigensystem *)
2087        TheIntermediateEigensystems[numEv_, E0pv_, E1pv_, E2pv_,
    E3pv_, ζv_] := (
2088          theNumericBlocks = (#[E0pv, E1pv, E2pv, E3pv, ζv]&) /@
    compiledDiagonal;
2089          theIntermediateEigensystems = Eigensystem /@
    theNumericBlocks;
2090          Js     = AllowedJ[numEv];
2091          basisJ = BasisLSJMJ[numEv, "AsAssociation"->True];
2092          (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
2093          elevatedIntermediateEigensystems = MapIndexed[EigenLever
    [#1, 2Js[[#2[[1]]]]+1 ]&, theIntermediateEigensystems];
2094          (* Identify a single MJ to keep *)
2095          pivot          = If[EvenQ[numEv],0,-1/2];
2096          LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
2097          (* calculate the multiplet assignments that the
    intermediate basis eigenvectors have *)
2098          needlePosition = 0;
2099          multipletAssignments = Table[
2100            (
2101              J         = Js[[idx]];
2102              eigenVecs = theIntermediateEigensystems[[idx]][[2]];
2103              majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
2104              majorComponentIndices       += needlePosition;
2105              needlePosition              += Length[
    majorComponentIndices];
2106              majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
2107              (* All of the degenerate eigenvectors belong to the
    same multiplet*)
2108              elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments, 2J+1];
2109              elevatedMultipletAssignments
2110            ),
2111          {idx, 1, Length[Js]}
2112          ];
2113
2114          (* put together the multiplet assignments and the energies
    *)
2115          freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
2116          freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
2117          (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
2118          basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
2119          basisChanger = SparseArray[basisChanger];
2120          Return[{theIntermediateEigensystems,
2121            multipletAssignments,
2122            elevatedIntermediateEigensystems,
2123            freeIenergiesAndMultiplets,
2124            basisChanger}]
2125      );
2126
2127      PrintFun["Calculating the intermediate eigensystems for ",ln,
    " using free-ion params from LaF3 ..."];
```

```
2128        (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
2129        {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
2130
2131        (* use that intermediate coupling basis to compile a function
     for the full Hamiltonian *)
2132        allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
2133        (* important that the intermediate coupling basis have
    attached energies, which make possible the truncation *)
2134        ordering = Ordering[allFreeEnergies];
2135        (* sort the free ion energies and determine which indices
    should be included in the truncation *)
2136        allFreeEnergiesSorted          = Sort[allFreeEnergies];
2137        {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
2138        (* determine the index at which the energy is equal or larger
     than the truncation energy *)
2139        sortedTruncationIndex = Which[
2140          truncationEnergy > (maxFreeEnergy-minFreeEnergy),
2141          hamDim,
2142          True,
2143          FirstPosition[allFreeEnergiesSorted - Min[
    allFreeEnergiesSorted],x_/;x>truncationEnergy,{0},1][[1]]
2144        ];
2145        (* the actual energy at which the truncation is made *)
2146        roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
2147
2148        (* the indices that participate in the truncation *)
2149        truncationIndices = ordering[[;;sortedTruncationIndex]];
2150        PrintFun["Computing the block structure of the change of
    basis array ..."];
2151        blockSizes         = BlockArrayDimensionsArray[ham];
2152        basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
2153        blockShifts        = First /@ Diagonal[blockSizes];
2154        numBlocks          = Length[blockSizes];
2155
2156        (* using the ham (with all the symbols) change the basis to
    the computed one *)
2157        PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
2158        intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks
    ];
2159        PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
2160        Do[
2161          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
2162          {rowIdx, 1, numBlocks},
2163          {colIdx, 1, numBlocks}
2164        ];
2165        intermediateHam = BlockMatrixMultiply[BlockTranspose[
    basisChangerBlocks],intermediateHam];
2166        PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
2167        Do[
2168          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
2169          {rowIdx, 1, numBlocks},
2170          {colIdx, 1, numBlocks}
2171        ];
2172        (* using the truncation indices truncate that one *)
2173        PrintFun["Truncating the Hamiltonian ..."];
2174        truncatedIntermediateHam = TruncateBlockArray[intermediateHam
    , truncationIndices, blockShifts];
2175        (* these are the basis vectors for the truncated hamiltonian
    *)
2176        PrintFun["Saving the truncated intermediate basis ..."];
2177        truncatedIntermediateBasis = basisChanger[[All,
    truncationIndices]];
2178
2179        PrintFun["Compiling a function for the truncated Hamiltonian
    ..."];
2180        (* compile a function that will calculate the truncated
```

```
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
2181        compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
    Evaluate[truncatedIntermediateHam]];
2182        (* save the compiled function *)
2183        PrintFun["Saving the compiled function for the truncated
    Hamiltonian and the truncated intermediate basis ..."];
2184        Export[compiledIntermediateFname, {
    compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
2185      )
2186    ];
2187    )
2188    ];
2189
2190    truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
2191    PrintFun["The truncated Hamiltonian has a dimension of ",
    truncationUmbral, "x", truncationUmbral, " ..."];
2192    presentDataIndices = Select[presentDataIndices, # <=
    truncationUmbral &];
2193    solCompendium["presentDataIndices"] = presentDataIndices;
2194
2195    (* the problemVars are the symbols that will be fitted for *)
2196
2197    PrintFun["Starting up the fitting process using the Levenberg-
    Marquardt method ..."];
2198    (* using the problemVars I need to create the argument list
    including _?NumericQ *)
2199    problemVarsQ       = (ToString[#] <> "_?NumericQ") & /@
    problemVars;
2200    problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
2201    (* we also need to have the padded arguments with the variables
    in the right position and the fixed values in the remaining ones
    *)
2202    problemVarsPositions = Position[allVars, #][[1, 1]] & /@
    problemVars;
2203    problemVarsString    = StringJoin[Riffle[ToString /@ problemVars,
     ", "]];
2204    (* to feed parameters to the Hamiltonian, which includes all
    parameters, we need to form the set of arguments, with fixed
    values where needed, and the variables in the right position *)
2205    varsWithConstants                          = standardValues;
2206    varsWithConstants[[problemVarsPositions]] = problemVars;
2207    varsWithConstantsString                    = ToString[
    varsWithConstants];
2208
2209    (* this following function serves eigenvalues from the
    Hamiltonian, has memoization so it might grow to use a lot of RAM
    *)
2210    Clear[HamSortedEigenvalues];
2211    hamEigenvaluesTemplate = StringTemplate["
2212    HamSortedEigenvalues[`problemVarsQ`]:=(
2213      ham          = compileIntermediateTruncatedHam@@`
    varsWithConstants`;
2214      eigenValues = Chop[Sort@Eigenvalues@ham];
2215      eigenValues = eigenValues - Min[eigenValues];
2216      HamSortedEigenvalues[`problemVarsString`] = eigenValues;
2217      Return[eigenValues]
2218    )"];
2219    hamString = hamEigenvaluesTemplate[<|
2220        "problemVarsQ"      -> problemVarsQString,
2221        "varsWithConstants" -> varsWithConstantsString,
2222        "problemVarsString" -> problemVarsString
2223        |>];
2224    ToExpression[hamString];
2225
2226    (* we also need a function that will pick the i-th eigenvalue,
    this seems unnecessary but it's needed to form the right
    functional form expected by the Levenberg-Marquardt method *)
2227    eigenvalueDispenserTemplate = StringTemplate["
2228    PartialHamEigenvalues[`problemVarsQ`][i_]:=(
2229      eigenVals = HamSortedEigenvalues[`problemVarsString`];
2230      eigenVals[[i]]
2231    )
2232    "];
2233    eigenValueDispenserString = eigenvalueDispenserTemplate[<|
2234        "problemVarsQ"      -> problemVarsQString,
```

```
2235          "problemVarsString" -> problemVarsString
2236          |>];
2237      ToExpression[eigenValueDispenserString];
2238
2239      PrintFun["Determining the free variables after constraints ..."];
2240      constrainedProblemVars     = (problemVars /. constraints);
2241      constrainedProblemVarsList = Variables[constrainedProblemVars];
2242      If[addShift,
2243        PrintFun["Adding a constant shift to the fitting parameters ...
          "];
2244        constrainedProblemVarsList = Append[constrainedProblemVarsList,
           \[Epsilon]]
2245      ];
2246
2247      indepVars = Complement[problemVars, #[[1]] & /@ constraints];
2248      stringPartialVars = ToString/@constrainedProblemVarsList;
2249
2250      paramSols  = {};
2251      rmsHistory = {};
2252      steps      = 0;
2253      problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
2254      If[addShift,
2255        problemVarsWithStartValues = Append[problemVarsWithStartValues,
           {\[Epsilon] ,0}];
2256      ];
2257      openNotebooks     = If[runningInteractive,
2258                ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
          [],
2259                {}];
2260      If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
          ProgressView"],
2261        ProgressNotebook["Basic"->False]
2262      ];
2263      degressOfFreedom = Length[presentDataIndices] - Length[
          problemVars] - 1;
2264      PrintFun["Fitting for ", Length[presentDataIndices], " data
          points with ", Length[problemVars], " free parameters.", " The
          effective degrees of freedom are ", degressOfFreedom, " ..."];
2265
2266      PrintFun["Fitting model to data ..."];
2267      startTime   = Now;
2268      shiftToggle = If[addShift, 1, 0];
2269      sol = FindMinimum[
2270        Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@
          constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
2271        {j, presentDataIndices}],
2272        problemVarsWithStartValues,
2273        Method        -> "LevenbergMarquardt",
2274        MaxIterations -> OptionValue["MaxIterations"],
2275        AccuracyGoal  -> OptionValue["AccuracyGoal"],
2276        StepMonitor :> (
2277          steps      += 1;
2278          currentSqSum = Sum[(expData[[j]][[1]] - (
          PartialHamEigenvalues @@ constrainedProblemVars)[j] - shiftToggle
          * \[Epsilon])^2, {j, presentDataIndices}];
2279          currentRMS = Sqrt[currentSqSum / degressOfFreedom];
2280          paramSols  = AddToList[paramSols, constrainedProblemVarsList,
           maxHistory];
2281          rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
2282          )
2283        ];
2284      endTime = Now;
2285      timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
2286      PrintFun["Solution found in ", timeTaken, "s"];
2287
2288      solVec = constrainedProblemVars /. sol[[-1]];
2289      indepSolVec = indepVars /. sol[[-1]];
2290      If[addShift,
2291        \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
2292        \[Epsilon]Best = 0
2293      ];
2294      fullSolVec = standardValues;
2295      fullSolVec[[problemVarsPositions]] = solVec;
2296      PrintFun["Calculating the truncated numerical Hamiltonian
          corresponding to the solution ..."];
2297      fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
```

```
PrintFun["Calculating energies and eigenvectors ..."];
{eigenEnergies, eigenVectors} = Eigensystem[fullHam];
states = Transpose[{eigenEnergies, eigenVectors}];
states = SortBy[states, First];
eigenEnergies = First /@ states;
PrintFun["Shifting energies to make ground state zero of energy ..."];
eigenEnergies = eigenEnergies - eigenEnergies[[1]];
PrintFun["Calculating the linear approximant to each eigenvalue ..."];
allVarsVec = Transpose[{allVars}];
p0 = Transpose[{fullSolVec}];
linMat = {};
If[addShift,
  tail = -2,
  tail = -1];
Do[
  (
    aVarPosition = Position[allVars, aVar][[1, 1]];
    isolationValues = ConstantArray[0, Length[allVars]];
    isolationValues[[aVarPosition]] = 1;
    dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[constraints]];
    Do[
      isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] = dVar[[2]],
      {dVar, dependentVars}
    ];
    perHam  = compileIntermediateTruncatedHam @@ isolationValues;
    lin     = FirstOrderPerturbation[Last /@ states, perHam];
    linMat  = Append[linMat, lin];
  ),
  {aVar, constrainedProblemVarsList[[;;tail]]}
];
PrintFun["Removing the gradient of the ground state ..."];
linMat = (# - #[[1]] & /@ linMat);
PrintFun["Transposing derivative matrices into columns ..."];
linMat  = Transpose[linMat];

PrintFun["Calculating the eigenvalue vector at solution ..."];
\[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
PrintFun["Putting together the experimental vector ..."];
\[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices]]}];
problemVarsVec = If[addShift,
  Transpose[{constrainedProblemVarsList[[;;-2]]}],
  Transpose[{constrainedProblemVarsList}]
];
indepSolVecVec = Transpose[{indepSolVec}];
PrintFun["Calculating the difference between eigenvalues at solution ..."];
diff = If[linMat == {},
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[presentDataIndices]].(problemVarsVec - indepSolVecVec)
];
PrintFun["Calculating the sum of squares of differences around solution ... "];
sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
PrintFun["Calculating the minimum (which should coincide with sol) ..."];
minpoly       = sqdiff /. AssociationThread[problemVars -> solVec];
fmSolAssoc    = Association[sol[[2]]];
If[\[Sigma]exp == Automatic,
  \[Sigma]exp = Sqrt[minpoly / degressOfFreedom];
];
\[CapitalDelta]\[Chi]2 = Sqrt[degressOfFreedom];
Amat = (1/\[Sigma]exp^2) * Transpose[linMat[[presentDataIndices]]].linMat[[presentDataIndices]];
paramIntervals = EllipsoidBoundingBox[Amat, \[CapitalDelta]\[Chi]2];
PrintFun["Calculating the uncertainty in the parameters ..."];
solWithUncertainty = Table[
  (
    aVar        = constrainedProblemVarsList[[varIdx]];
```

```
        paramBest   = aVar /. fmSolAssoc;
        (aVar -> {paramBest, paramIntervals[[varIdx, 2]]})
      ),
    {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
    ];

    bestRMS    = Sqrt[minpoly / degressOfFreedom];
    bestParams = sol[[2]];
    bestWithConstraints = Association@Join[constraints, bestParams];
    bestWithConstraints = bestWithConstraints /. bestWithConstraints;
    bestWithConstraints = (# + 0.) & /@ bestWithConstraints;

    solCompendium["degreesOfFreedom"]    = degressOfFreedom;
    solCompendium["solWithUncertainty"]  = solWithUncertainty;
    solCompendium["truncatedDim"]        = truncationUmbral;
    solCompendium["fittedLevels"]        = Length[presentDataIndices
    ];
    solCompendium["actualSteps"]         = steps;
    solCompendium["bestRMS"]             = bestRMS;
    solCompendium["problemVars"]         = problemVars;
    solCompendium["paramSols"]           = paramSols;
    solCompendium["rmsHistory"]          = rmsHistory;
    solCompendium["Appendix"]            = OptionValue["
    AppendToLogFile"];
    solCompendium["timeTaken/s"]         = timeTaken;
    solCompendium["bestParams"]          = bestParams;
    solCompendium["bestParamsWithConstraints"] = bestWithConstraints;

    If[OptionValue["SaveEigenvectors"],
        solCompendium["states"] = {#[[1]] + \[Epsilon]Best, #[[2]]}
    &/@ (Chop /@ ShiftedLevels[states]),
        (
          finalEnergies = Sort[First /@ states];
          finalEnergies = finalEnergies - finalEnergies[[1]];
          finalEnergies = finalEnergies + \[Epsilon]Best;
          finalEnergies  = Chop /@ finalEnergies;
          solCompendium["energies"] = finalEnergies;
      )
    ];
    If[OptionValue["SaveToLog"],
      PrintFun["Saving the solution to the log file ..."];
      LogSol[solCompendium, logFilePrefix];
    ];
    PrintFun["Finished ..."];
    Return[solCompendium];
  )
];

StringToSLJ[string_] := Module[
  {stringed = string, LS, J, LSindex},
  (
  If[StringContainsQ[stringed, "+"],
   Return["mixed"]
   ];
  LS = StringTake[stringed, {1, 2}];
  If[StringContainsQ[stringed, "("],
   (
     LSindex =
     StringCases[stringed, RegularExpression["\\(([^)]*)\\)"] :> "$1"
   ];
    LS = LS <> LSindex;
    stringed = StringSplit[stringed, ")"][[-1]];
    J = ToExpression[stringed];
    ),
   (
    J = ToExpression@StringTake[stringed, {3, -1}];
    )
   ];
  {LS, J}
  )
];

FreeIonSolver::usage="This function takes a list of experimental data
     and the number of electrons in the lanthanide ion and returns the
     free-ion parameters that best fit the data. The options are:
  - F4F6_SlaterRatios: a list of two numbers that represent the ratio
```

```
    of F4 to F2 and F6 to F2, respectively.
  - PrintFun: a function that will be used to print the progress of the fitting process.
  - MaxIterations: the maximum number of iterations that the fitting process will run.
  - MaxMultiplets: the maximum number of multiplets that will be used in the fitting process.
  - MaxPercent: the maximum percentage of the data that can be off by the fitting.
  - SubSetBounds: a list of two numbers that represent the minimum and maximum number of multiplets that will be used in the fitting process.
The function returns an association with the following keys:
  - bestParams: the best parameters found in the fitting.
  - worstRelativeError: the worst relative error in the fitting.
  - SlaterRatios: the Slater ratios used in the fitting.
  - usedBaricenters: the baricenters used in the fitting.
If no acceptable solution is found, the function will return all solutions that are not worse than 10*MaxPercent. A solution is acceptable if the worst relative error is less than the MaxPercent option.
";
Options[FreeIonSolver] = {
  "F4F6_SlaterRatios" -> {0.707, 0.516},
  "PrintFun" -> PrintTemporary,
  "MaxIterations" -> 10000,
  "MaxMultiplets" -> 12,
  "MaxPercent" -> 3.,
  "SubSetBounds" -> {5, 12}
  };
FreeIonSolver[expData_, numE_, OptionsPattern[]] := Module[
  {maxMultiplets, maxPercent, F4overF2, F6overF2, PrintFun,
    minSubSetSize, maxSubSetSize, multipletEnergies, numMultiplets,
    allEqns, subsetSizes, ln, solutions, subsets, subset, eqns, slope,
     intercept, meritFun, sol, goodThings, bestThings, bestOfAll,
    finalSol, usedMultiplets, usedBaricenters},
  (
    maxMultiplets = OptionValue["MaxMultiplets"];
    maxIterations = OptionValue["MaxIterations"];
    maxPercent    = OptionValue["MaxPercent"];
    F4overF2      = OptionValue["F4F6_SlaterRatios"][[1]];
    F6overF2      = OptionValue["F4F6_SlaterRatios"][[2]];
    PrintFun      = OptionValue["PrintFun"];
    minSubSetSize = OptionValue["SubSetBounds"][[1]];
    maxSubSetSize = OptionValue["SubSetBounds"][[2]];
    freeIonParams = {F0, F2, F4, F6, ζ};
    ln            = theLanthanides[[numE]];

    PrintFun["Parsing the barycenters of the different multiplets ..."];
    multipletEnergies = Map[First, #] & /@ GroupBy[expData, #[[2]] &];
    multipletEnergies = Mean[Select[#, NumberQ]] & /@ multipletEnergies;
    multipletEnergies = Select[multipletEnergies, FreeQ[#, Mean] &];
    multipletEnergies = KeySelect[KeyMap[StringToSLJ, multipletEnergies], # =!= "mixed" &];
    multipletEnergies = KeyMap[Prepend[#, numE] &, multipletEnergies];
    numMultiplets     = Length[multipletEnergies];

    PrintFun["Composing the system of equations for the free-ion energies ..."];
    allEqns     = KeyValueMap[FreeIonTable[#1] == #2 &, multipletEnergies];
    allEqns     = Select[allEqns, FreeQ[#, Missing] &];
    allEqns     = Append[Coefficient[#[[1]], {F0, F2, F4, F6, ζ}], #[[2]]] & /@ allEqns;
    allEqns     = allEqns[[;; Min[Length[allEqns], maxMultiplets]]];
    subsetSizes = Range[minSubSetSize, Min[numMultiplets, maxSubSetSize]];
    numSubsets = {#, Binomial[numMultiplets, #]} & /@ subsetSizes;
    numSubsets = Transpose@SortBy[numSubsets, Last];
    accSizes = Accumulate[numSubsets[[2]]];
    numSubsets = Transpose@Append[numSubsets, accSizes];
    lastSub = SelectFirst[numSubsets, #[[3]] > 1000 &, Last[
```

```
    numSubsets]];
    lastPosition = Position[numSubsets, lastSub][[1, 1]];
    chosenSubsetSizes = #[[1]] & /@ numSubsets[[;; lastPosition]];
    solutions   = <||>;

    PrintFun["Selecting subsets of different lengths and fitting with
     ratio-constraints ..."];
    Do[
      subsets = Subsets[Range[1, Length[allEqns]], {subsetSize}];
      PrintFun["Considering ", Length[subsets], " barycenter subsets
    of size ", subsetSize, " ..."];
      Do[
      (
        subset    = subsets[[subsetIndex]];
        eqns      = allEqns[[subset]];
        slope     = #[[;; 5]] & /@ eqns;
        intercept = #[[6]] & /@ eqns;
        meritFun = Max[100 * Expand[Abs[(slope . freeIonParams -
    intercept)]/intercept]];
        sol = NMinimize[{meritFun,
          F0 > 0,
          F2 > 1000.,
          F4 == F4overF2 * F2,
          F6 == F6overF2 * F2,
          ζ > 0},
          freeIonParams,
          MaxIterations -> maxIterations,
          Method -> "Convex"];
        solutions[{subsetSize, subset}] = sol;
        )
      , {subsetIndex, 1, Length[subsets]}
      ],
      {subsetSize, chosenSubsetSizes}
    ];

    PrintFun["Collecting solutions of different subset size ..."];
    goodThings = Table[
      (chunk =
        Normal[Sort[
          KeySelect[#[[1]] & /@ solutions, #[[1]] == subSize &]]];
        If[Length[chunk] >= 1,
        chunk[[1]],
        Null
        ]), {subSize, subsetSizes}];
    goodThings = Select[goodThings, Head[#] === Rule &];

    PrintFun["Picking the solutions that are not worse than ",
    maxPercent, "% ..."];
    bestThings = Select[goodThings, #[[2]] <= maxPercent &];
    If[bestThings == {},
      Print["No acceptable solution found, consider increasing
    maxPercent or inspecting the given data ..."];
      Return[goodThings];
    ];

    PrintFun["Keeping the solution with the largest number of used
    barycenters ..."];
    bestOfAll = bestThings[[-1]];
    sol       = solutions[bestOfAll[[1]]];
    subset    = bestOfAll[[1, 2]];
    eqns      = allEqns[[subset]];
    slope     = #[[;; 5]] & /@ eqns;
    intercept = #[[6]] & /@ eqns;
    usedMultiplets  = Keys[multipletEnergies][[subset]];
    usedBaricenters = {#, multipletEnergies[#]} & /@ usedMultiplets;
    uniqeLS = DeleteDuplicates[#[[2]] &/@ Keys[multipletEnergies]];
    solAssoc = Association[sol[[2]]];
    usedLaF3 = False;
    If[Length[uniqeLS] == 1,
      (
        Print["There is too little data to find Slater parameters,
    using the ones for LaF3, and keeping the fitted spin-orbit zeta
    ..."];
        laf3params = LoadLaF3Parameters[ln];
        usedLaF3   = True;
        solAssoc[F0] = laf3params[F0];
```

```
        solAssoc[F2] = laf3params[F2];
        solAssoc[F4] = laf3params[F4];
        solAssoc[F6] = laf3params[F6];
      )
    ];
    finalSol = <| "bestParams" -> solAssoc,
      "usedLaF3" -> usedLaF3,
      "worstRelativeError" -> sol[[1]],
      "SlaterRatios" -> {F4overF2, F6overF2},
      "usedBaricenters" -> usedBaricenters|>;
    Return[finalSol];
)
];

EndPackage[];
```

18.3 qplotter.m

This module has a few useful plotting routines.

```

BeginPackage["qplotter`"];

GetColor;
IndexMappingPlot;
ListLabelPlot;
AutoGraphicsGrid;
SpectrumPlot;
WaveToRGB;

Begin["`Private`"];

  AutoGraphicsGrid::usage="AutoGraphicsGrid[graphsList] takes a list
    of graphics and creates a GraphicsGrid with them. The number of
    columns and rows is chosen automatically so that the grid has a
    squarish shape.";
  Options[AutoGraphicsGrid] = Options[GraphicsGrid];
  AutoGraphicsGrid[graphsList_, opts : OptionsPattern[]] :=
    (
      numGraphs = Length[graphsList];
      width = Floor[Sqrt[numGraphs]];
      height = Ceiling[numGraphs/width];
      groupedGraphs = Partition[graphsList, width, width, 1, Null];
      GraphicsGrid[groupedGraphs, opts]
    )

  Options[IndexMappingPlot] = Options[Graphics];
  IndexMappingPlot::usage =
    "IndexMappingPlot[pairs] take a list of pairs of integers and
    creates a visual representation of how they are paired. The first
    indices being depicted in the bottom and the second indices being
    depicted on top.";
  IndexMappingPlot[pairs_, opts : OptionsPattern[]] := Module[{width,
     height}, (
    width = Max[First /@ pairs];
    height = width/3;
    Return[
      Graphics[{{Tooltip[Point[{#[[1]], 0}],#[[1]]]}, Tooltip[Point
    [{#[[2]], height}],#[[2]]],
          Line[{{#[[1]], 0}, {#[[2]], height}}]} & /@ pairs, opts,
    ImageSize -> 800]]
    )
    ]

  TickCompressor[fTicks_] :=
  Module[{avgTicks, prevTickLabel, groupCounter, groupTally, idx,
    tickPosition, tickLabel, avgPosition, groupLabel}, (avgTicks =
    {};
    prevTickLabel = fTicks[[1, 2]];
    groupCounter = 0;
    groupTally = 0;
    idx = 1;
    Do[({tickPosition, tickLabel} = tick;
      If[
      tickLabel === prevTickLabel,
      (groupCounter += 1;
```

```
        groupTally += tickPosition;
        groupLabel = tickLabel;),
      (
        avgPosition = groupTally/groupCounter;
        avgTicks = Append[avgTicks, {avgPosition, groupLabel}];
        groupCounter = 1;
        groupTally = tickPosition;
        groupLabel = tickLabel;
        )
      ];
      If[idx != Length[fTicks],
      prevTickLabel = tickLabel;
      idx += 1;]
      ), {tick, fTicks}];
    If[Or[Not[prevTickLabel === tickLabel], groupCounter > 1],
    (
      avgPosition = groupTally/groupCounter;
      avgTicks = Append[avgTicks, {avgPosition, groupLabel}];
      )
    ];
    Return[avgTicks];)]

GetColor[s_Style] := s /. Style[_, c_] :> c
GetColor[_] := Black

ListLabelPlot::usage="ListLabelPlot[data, labels] takes a list of
  numbers with corresponding labels. The data is grouped according
  to the labels and a ListPlot is created with them so that each
  group has a different color and their corresponding label is shown
   in the horizontal axis.";
Options[ListLabelPlot] = Join[Options[ListPlot], {"TickCompression"
  ->True,
"LabelLevels"->1}];
ListLabelPlot[data_, labels_, opts : OptionsPattern[]] := Module[
  {uniqueLabels, pallete, groupedByTerm, groupedKeys, scatterGroups
  ,
  groupedColors, frameTicks, compTicks, bottomTicks, topTicks},
  (
  uniqueLabels  = DeleteDuplicates[labels];
  pallete = Table[ColorData["Rainbow", i], {i, 0, 1,
      1/(Length[uniqueLabels] - 1)}];
  uniqueLabels  = (#[[1]] -> #[[2]]) & /@ Transpose[{RandomSample[
  uniqueLabels], pallete}];
  uniqueLabels  = Association[uniqueLabels];
  groupedByTerm = GroupBy[Transpose[{labels, Range[Length[data]],
  data}], First];
  groupedKeys   = Keys[groupedByTerm];
  scatterGroups = Transpose[Transpose[#][[2 ;; 3]]] & /@ Values[
  groupedByTerm];
  groupedColors = uniqueLabels[#] & /@ groupedKeys;
  frameTicks    = {Transpose[{Range[Length[data]],
    Style[Rotate[#, 90 Degree], uniqueLabels[#]] & /@ labels}],
    Automatic};
    If[OptionValue["TickCompression"], (
        compTicks = TickCompressor[frameTicks[[1]]];
        bottomTicks =
            MapIndexed[
            If[EvenQ[First[#2]], {#1[[1]],
                Tooltip[Style["\[SmallCircle]", GetColor
  [#1[[2]]]],#1[[2]]]
                }, #1] &, compTicks];
        topTicks =
            MapIndexed[
            If[OddQ[First[#2]], {#1[[1]],
                Tooltip[Style["\[SmallCircle]", GetColor
  [#1[[2]]]],#1[[2]]]
                }, #1] &, compTicks];
        frameTicks = {{Automatic, Automatic}, {bottomTicks,
  topTicks}};)
  ];
  ListPlot[scatterGroups,
    opts,
    Frame       -> True,
    AxesStyle   -> {Directive[Black, Dotted], Automatic},
    PlotStyle   -> groupedColors,
    FrameTicks -> frameTicks]
```

```
    )
    ]

  WaveToRGB::usage="WaveToRGB[wave, gamma] takes a wavelength in nm
    and returns the corresponding RGB color. The gamma parameter is
    optional and defaults to 0.8. The wavelength wave is assumed to be
     in nm. If the wavelength is below 380 the color will be the same
    as for 380 nm. If the wavelength is above 750 the color will be
    the same as for 750 nm. The function returns an RGBColor object.
    REF: https://www.noah.org/wiki/wave_to_rgb_in_Python. ";
  WaveToRGB[wave_, gamma_ : 0.8] := (
    wavelength = (wave);
    Which[
      wavelength < 380,
        wavelength = 380,
      wavelength > 750,
        wavelength = 750
    ];
    Which[380 <= wavelength <= 440,
    (
      attenuation = 0.3 + 0.7*(wavelength - 380)/(440 - 380);
      R = ((-(wavelength - 440)/(440 - 380))*attenuation)^gamma;
      G = 0.0;
      B = (1.0*attenuation)^gamma;
      ),
    440 <= wavelength <= 490,
    (
      R = 0.0;
      G = ((wavelength - 440)/(490 - 440))^gamma;
      B = 1.0;
      ),
    490 <= wavelength <= 510,
    (
      R = 0.0;
      G = 1.0;
      B = (-(wavelength - 510)/(510 - 490))^gamma;
      ),
    510 <= wavelength <= 580,
    (
      R = ((wavelength - 510)/(580 - 510))^gamma;
      G = 1.0;
      B = 0.0;
      ),
    580 <= wavelength <= 645,
    (
      R = 1.0;
      G = (-(wavelength - 645)/(645 - 580))^gamma;
      B = 0.0;
      ),
    645 <= wavelength <= 750,
    (
      attenuation = 0.3 + 0.7*(750 - wavelength)/(750 - 645);
      R = (1.0*attenuation)^gamma;
      G = 0.0;
      B = 0.0;
      ),
    True,
    (
      R = 0;
      G = 0;
      B = 0;
      )];
    Return[RGBColor[R, G, B]]
    )

  FuzzyRectangle::usage = "FuzzyRectangle[xCenter, width, ymin,
    height, color] creates a polygon with a fuzzy edge. The polygon is
     centered at xCenter and has a full horizontal width of width. The
     bottom of the polygon is at ymin and the height is height. The
    color of the polygon is color. The left edge and the right edge of
     the resulting polygon will be transparent and the middle will be
    colored. The polygon is returned as a list of polygons.";
  FuzzyRectangle[xCenter_, width_, ymin_, height_, color_, intensity_
    :1] := Module[
    {intenseColor, nocolor, ymax, polys},
  (
```

```
    nocolor = Directive[Opacity[0], color];
    ymax = ymin + height;
    intenseColor = Directive[Opacity[intensity], color];
    polys = {
      Polygon[{
        {xCenter - width/2, ymin},
        {xCenter, ymin},
        {xCenter, ymax},
        {xCenter - width/2, ymax}},
        VertexColors -> {
          nocolor,
          intenseColor,
          intenseColor,
          nocolor,
          nocolor}],
      Polygon[{
        {xCenter, ymin},
        {xCenter + width/2, ymin},
        {xCenter + width/2, ymax},
        {xCenter, ymax}},
        VertexColors -> {
          intenseColor,
          nocolor,
          nocolor,
          intenseColor,
          intenseColor}]
      };
    Return[polys]
    );
]

Options[SpectrumPlot] = Options[Graphics];
Options[SpectrumPlot] = Join[Options[SpectrumPlot], {"Intensities"
  -> {},"Tooltips" -> True, "Comments" -> {}, "SpectrumFunction" ->
  WaveToRGB}];
SpectrumPlot::usage="SpectrumPlot[lines, widthToHeightAspect,
  lineWidth] takes a list of spectral lines and creates a visual
  representation of them. The lines are represented as fuzzy
  rectangles with a width of lineWidth and a height that is
  determined by the overall condition that the width to height ratio
   of the resulting graph is widthToHeightAspect. The color of the
  lines is determined by the wavelength of the line. The function
  assumes that the lines are given in nm.
If the lineWidth parameter is a single number, then every line
  shares that width. If the lineWidth parameter is a list of numbers
  , then each line has a different width. The function returns a
  Graphics object. The function also accepts any options that
  Graphics accepts. The background of the plot is black by default.
  The plot range is set to the minimum and maximum wavelength of the
   given lines.
Besides the options for Graphics the function also admits the
  option Intensities. This option is a list of numbers that
  determines the intensity of each line. If the Intensities option
  is not given, then the lines are drawn with full intensity. If the
   Intensities option is given, then the lines are drawn with the
  given intensity. The intensity is a number between 0 and 1.
The function also admits the option \"Tooltips\". If this option is
   set to True, then the lines will have a tooltip that shows the
  wavelength of the line. If this option is set to False, then the
  lines will not have a tooltip. The default value for this option
  is True.
If \"Tooltips\" is set to True and the option \"Comments\" is a non
  -empty list, then the tooltip will append the wavelength and the
  values in the comments list for the tooltips.
The function also admits the option \"SpectrumFunction\". This
  option is a function that takes a wavelength and returns a color.
  The default value for this option is WaveToRGB.
";
SpectrumPlot[lines_, widthToHeightAspect_, lineWidth_, opts :
  OptionsPattern[]] := Module[
  {minWave, maxWave, height, fuzzyLines},
(
  colorFun = OptionValue["SpectrumFunction"];
  {minWave, maxWave} = MinMax[lines];
  height     = (maxWave - minWave)/widthToHeightAspect;
  fuzzyLines = Which[
```

```
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] == 0,
        FuzzyRectangle[#, lineWidth, 0, height, colorFun[#]] &/@ lines,
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] == 0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1]] &, {lines, lineWidth}],
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] > 0,
        MapThread[FuzzyRectangle[#1, lineWidth, 0, height, colorFun[#1], #2] &, {lines, OptionValue["Intensities"]}],
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] > 0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1], #3] &, {lines, lineWidth, OptionValue["Intensities"]}]
    ];
    comments = Which[
      Length[OptionValue["Comments"]] > 0,
        MapThread[StringJoin[ToString[#1]<>" nm","\n",ToString[#2]]&, {lines, OptionValue["Comments"]}],
      Length[OptionValue["Comments"]] == 0,
        ToString[#] <>" nm" & /@ lines,
      True,
      {}
    ];
    If[OptionValue["Tooltips"],
      fuzzyLines = MapThread[Tooltip[#1, #2] &, {fuzzyLines, comments}];
    ];
    graphicsOpts = FilterRules[{opts}, Options[Graphics]];
    Graphics[fuzzyLines,
        graphicsOpts,
        Background -> Black,
        PlotRange -> {{minWave, maxWave}, {0, height}}]
    )
  ];

End[];

EndPackage[];
```

18.4 misc.m

This module includes a few functions useful for data-handling.

```
BeginPackage["misc`"];
(* Needs["MaTeX`"]; *)

ArrayBlocker;
BlockAndIndex;
BlockArrayDimensionsArray;
BlockMatrixMultiply;
BlockTranspose;

EllipsoidBoundingBox;
ExportToH5;
FirstOrderPerturbation;
FlattenBasis;

GetModificationDate;
HamTeX;
HelperNotebook;

RecoverBasis;
RemoveTrailingDigits;
ReplaceDiagonal;
RobustMissingQ;

RoundToSignificantFigures;
RoundValueWithUncertainty;
SecondOrderPerturbation;
SuperIdentity;

TextBasedProgressBar;
ToPythonSparseFunction;
ToPythonSymPyExpression;
TruncateBlockArray;
```

```
33
34 Begin["`Private`"];
35
36   ExtractSymbolNames;
37
38   RemoveTrailingDigits[s_String] := StringReplace[s,
       RegularExpression["\\d+$"] -> ""];
39
40   BlockTranspose::usage="BlcockTranspose[anArray] takes a 2D array
       with a congruent block structure and returns the transposed array
       with the same block structure.";
41   BlockTranspose[anArray_]:=(
42     Map[Transpose, Transpose[anArray], {2}]
43   );
44
45   BlockMatrixMultiply::usage="BlockMatrixMultiply[A,B] gives the
       matrix multiplication of A and B, with A and B having a compatible
        block structure that allows for matrix multiplication into a
       congruent block structure.";
46   BlockMatrixMultiply[Amat_,Bmat_]:=Module[{rowIdx,colIdx,sumIdx},
47   (
48     Table[
49       Sum[Amat[[rowIdx,sumIdx]].Bmat[[sumIdx,colIdx]],{sumIdx,1,
       Dimensions[Amat][[2]]}],
50     {rowIdx,1,Dimensions[Amat][[1]]},
51     {colIdx,1,Dimensions[Bmat][[2]]}
52     ]
53   )
54   ];
55
56   BlockAndIndex::usage="BlockAndIndex[{w1, w2, ...}, idx] takes a
       list of block lengths wi and an index idx. The function returns in
        which block idx would be in a a list defined by {Range[1,w1],
       Range[1+w1,w1+w2], ...}. The function also returns the position
       within the bin in which the given index would be found in. The
       function returns these two numbers as a list of two elements {
       blockIndex, blockSubIndex}.";
57   BlockAndIndex[blockSizes_List, index_Integer]:=Module[{
       accumulatedBlockSize,blockIndex, blockSubIndex},
58   (
59     accumulatedBlockSize = Accumulate[blockSizes];
60     If[accumulatedBlockSize[[-1]]-index<0,
61       Print["Index out of bounds"];
62       Abort[]
63     ];
64     blockIndex    = Flatten[Position[accumulatedBlockSize-index,n_ /;
        n>=0]][[1]];
65     blockSubIndex = Mod[index-accumulatedBlockSize[[blockIndex]],
       blockSizes[[blockIndex]],1];
66     Return[{blockIndex,blockSubIndex}]
67   )
68   ];
69
70   TruncateBlockArray::usage="TruncateBlockArray[blockArray,
       truncationIndices, blockWidths] takes an array of blocks and
       selects the columns and rows corresponding to truncationIndices.
       The indices being given in what would be the ArrayFlatten[
       blockArray] version of the array. The blocks in the given array
       may be SparseArray. This is equivalent to FlattenArray[blockArray
       ][truncationIndices, truncationIndices] but may be more efficient
       if blockArray is sparse.";
71   TruncateBlockArray[blockArray_,truncationIndices_,blockWidths_]:=
       Module[
72   {truncatedArray,blockCol,blockRow,blockSubCol,blockSubRow},(
73   truncatedArray = Table[
74     {blockCol,blockSubCol} = BlockAndIndex[blockWidths,fullColIndex];
75     {blockRow,blockSubRow} = BlockAndIndex[blockWidths,fullRowIndex];
76     blockArray[[blockRow,blockCol]][[blockSubRow,blockSubCol]],
77   {fullRowIndex,truncationIndices},
78   {fullColIndex,truncationIndices}
79   ];
80   Return[truncatedArray]
81   )
82   ];
83
84   BlockArrayDimensionsArray::usage="BlockArrayDimensionsArray[
```

```
  blockArray] returns the array of block sizes in a given blocked
  array.";
BlockArrayDimensionsArray[blockArray_]:=(
  Map[Dimensions,blockArray,{2}]
);

ArrayBlocker::usage="ArrayBlocker[anArray, blockSizes] takes a flat
   2d array and a congruent 2D array of block sizes, and with them
  it returns the original array with the block structure imposed by
  blockSizes. The resulting array satisfies ArrayFlatten[
  blockedArray] == anArray, and also Map[Dimensions, blockedArray
  ,{2}] == blockSizes.";
ArrayBlocker[anArray_, blockSizes_] := Module[{rowStart,colStart,
  colEnd,numBlocks,blockedArray,blockSize,rowEnd,aBlock,idxRow,
  idxCol},(
   rowStart  = 1;
   colStart  = 1;
   colEnd    = 1;
   case      = Length[Dimensions[blockSizes]];
   Which[
     case == 3,
     (
       numBlocks = Length[blockSizes];
       blockedArray = Table[(
         blockSize = blockSizes[[idxRow,idxCol]];
         rowEnd    = rowStart+blockSize[[1]]-1;
         colEnd    = colStart+blockSize[[2]]-1;
         aBlock    = anArray[[rowStart;;rowEnd,colStart;;colEnd]];
         colStart  = colEnd+1;
         If[idxCol==numBlocks,
           rowStart=rowEnd+1;
           colStart=1;
         ];
         aBlock
     ),
     {idxRow,1,numBlocks},
     {idxCol,1,numBlocks}];
     ),
     case == 1,
     (
       expandedSizes = Table[
         {blockSizes[[idxRow]], blockSizes[[idxCol]]},
         {idxRow,1,Length[blockSizes]},
         {idxCol,1,Length[blockSizes]}
         ];
       numBlocks = Length[expandedSizes];
       blockedArray = Table[(
         blockSize = expandedSizes[[idxRow,idxCol]];
         rowEnd    = rowStart+blockSize[[1]]-1;
         colEnd    = colStart+blockSize[[2]]-1;
         aBlock    = anArray[[rowStart;;rowEnd,colStart;;colEnd]];
         colStart  = colEnd+1;
         If[idxCol==numBlocks,
           rowStart=rowEnd+1;
           colStart=1;
         ];
         aBlock
     ),
     {idxRow,1,numBlocks},
     {idxCol,1,numBlocks}];
     )
   ];
   Return[blockedArray]
)
];

ReplaceDiagonal::usage =
  "ReplaceDiagonal[matrix, repValue] replaces all the diagonal of
  the given array to the given value. The array is assumed to be
  square and the replacement value is assumed to be a number. The
  returned value is the array with the diagonal replaced. This
  function is useful for setting the diagonal of an array to a given
   value. The original array is not modified. The given array may be
   sparse.";
ReplaceDiagonal[matrix_, repValue_] :=
  ReplacePart[matrix,
```

```
      Table[{i, i} -> repValue, {i, 1, Length[matrix]}]];

Options[RoundValueWithUncertainty] = {"SetPrecision" -> False};
RoundValueWithUncertainty::usage = "RoundValueWithUncertainty[x,dx]
   given a number x together with an uncertainty dx this function
  rounds x to the first significant figure of dx and also rounds dx
  to have a single significant figure.
The returned value is a list with the form {roundedX, roundedDx}.
The option \"SetPrecision\" can be used to control whether the
  Mathematica precision of x and dx is also set accordingly to these
   rules, otherwise the rounded numbers still have the original
  precision of the input values.
If the position of the first significant figure of x is after the
  position of the first significant figure of dx, the function
  returns {0,dx} with dx rounded to one significant figure.";
RoundValueWithUncertainty[x_, dx_, OptionsPattern[]] := Module[
  {xExpo, dxExpo, sigFigs, roundedX, roundedDx, returning},
  (
    xExpo   = RealDigits[x][[2]];
    dxExpo  = RealDigits[dx][[2]];
    sigFigs = (xExpo - dxExpo) + 1;
    {roundedX, roundedDx} = If[sigFigs <= 0,
      {0., N@RoundToSignificantFigures[dx, 1]},
      N[
      {
        RoundToSignificantFigures[x, xExpo - dxExpo + 1],
        RoundToSignificantFigures[dx, 1]}
      ]
      ];
    returning = If[
      OptionValue["SetPrecision"],
      {SetPrecision[roundedX, Max[1, sigFigs]],
      SetPrecision[roundedDx, 1]},
      {roundedX, roundedDx}
      ];
    Return[returning]
    )
  ];

RoundToSignificantFigures::usage =
  "RoundToSignificantFigures[x, sigFigs] rounds x so that it only
  has \
sigFigs significant figures.";
RoundToSignificantFigures[x_, sigFigs_] :=
  Sign[x]*N[FromDigits[RealDigits[x, 10, sigFigs]]];

RobustMissingQ[expr_] := (FreeQ[expr, _Missing] === False);

TextBasedProgressBar[progress_, totalIterations_, prefix_:""] :=
  Module[
    {progMessage},
    progMessage = ToString[progress] <> "/" <> ToString[
  totalIterations];
    If[progress < totalIterations,
        WriteString["stdout", StringJoin[prefix, progMessage, "\r"
  ]],
        WriteString["stdout", StringJoin[prefix, progMessage, "\n"
  ]]
    ];
];

FirstOrderPerturbation::usage="Given the eigenVectors of a matrix A
   (which doesn't need to be given) together with a corresponding
  perturbation matrix perMatrix, this function calculates the first
  derivative of the eigenvalues with respect to the scale factor of
  the perturbation matrix. In the sense that the eigenvalues of the
  matrix A + \[Beta] perMatrix are to first order equal to \[Lambda]_i +
  \[Delta]_i \[Beta], where the \[Delta]_i are the returned values. This
  assuming that the eigenvalues are non-degenerate.";
FirstOrderPerturbation[eigenVectors_,
  perMatrix_] := (Chop@Diagonal[
  Conjugate@eigenVectors . perMatrix . Transpose[eigenVectors]])

SecondOrderPerturbation::usage="Given the eigenValues and
  eigenVectors of a matrix A (which doesn't need to be given)
  together with a corresponding perturbation matrix perMatrix, this
```

```
    function calculates the second derivative of the eigenvalues with
    respect to the scale factor of the perturbation matrix. In the
    sense that the eigenvalues of the matrix A + β perMatrix are to
    second order equal to \[Lambda] + \[Delta]_i β + \[Delta]_i^{(2)
    }/2 β^2, where the \[Delta]_i^{(2)} are the returned values. The
    eigenvalues and eigenvectors are assumed to be given in the same
    order, i.e. the ith eigenvalue corresponds to the ith eigenvector.
     This assuming that the eigenvalues are non-degenerate.";
200 SecondOrderPerturbation[eigenValues_, eigenVectors_, perMatrix_] :=
     (
201   dim = Length[perMatrix];
202   eigenBras = Conjugate[eigenVectors];
203   eigenKets = eigenVectors;
204   matV = Abs[eigenBras . perMatrix . Transpose[eigenKets]]^2;
205   OneOver[x_, y_] := If[x == y, 0, 1/(x - y)];
206   eigenDiffs = Outer[OneOver, eigenValues, eigenValues, 1];
207   pProduct = Transpose[eigenDiffs]*matV;
208   Return[2*(Total /@ Transpose[pProduct])];
209   )
210
211 SuperIdentity::usage="SuperIdentity[args] returns the arguments
    passed to it. This is useful for defining a function that does
    nothing, but that can be used in a composition.";
212 SuperIdentity[args___] := {args};
213
214 ExtractSymbolNames[expr_Hold] := Module[
215   {strSymbols},
216   strSymbols = ToString[expr, InputForm];
217   StringCases[strSymbols, RegularExpression["\\w+"]][[2 ;;]]
218   ]
219
220 ExportToH5::usage =
221   "ExportToH5[fname, Hold[{symbol1, symbol2, ...}]] takes an .h5
    filename and a held list of symbols and export to the .h5 file the
     values of the symbols with keys equal the symbol names. The
    values of the symbols cannot be arbitrary, for instance a list
    which mixes numbers and strings will fail, but an Association with
     mixed values exports ok. Do give it a try.
    If the file is already present in disk, this function will
    overwrite it by default. If the value of a given symbol contains
    symbolic numbers, e.g. \[Pi], these will be converted to floats in
     the exported file.";
223 Options[ExportToH5] = {"Overwrite" -> True};
224 ExportToH5[fname_String, symbols_Hold, OptionsPattern[]] := (
225   If[And[FileExistsQ[fname], OptionValue["Overwrite"]],
226   (
227     Print["File already exists, overwriting ..."];
228     DeleteFile[fname];
229     )
230   ];
231   symbolNames = ExtractSymbolNames[symbols];
232   Do[(Print[symbolName];
233     Export[fname, ToExpression[symbolName], {"Datasets", symbolName
    },
234     OverwriteTarget -> "Append"]
235     ), {symbolName, symbolNames}]
236   )
237
238 HelperNotebook::usage="HelperNotebook[nbName] creates a separate
    notebook and returns a function that can be used to print to the
    bottom of it. The name of the notebook, nbName, is optional and
    defaults to OUT.";
239 HelperNotebook[nbName_:"OUT"] :=
240 Module[{screenDims, screenWidth, screenHeight, nbWidth, leftMargin,
    PrintToOutputNb}, (
241   screenDims =
242     SystemInformation["Devices", "ScreenInformation"][[1, 2, 2]];
243   screenWidth = screenDims[[1, 2]];
244   screenHeight = screenDims[[2, 2]];
245   nbWidth = Round[screenWidth/3];
246   leftMargin = screenWidth - nbWidth;
247   outputNb = CreateDocument[{}, WindowTitle -> nbName,
248     WindowMargins -> {{leftMargin, Automatic}, {Automatic,
249         Automatic}}, WindowSize -> {nbWidth, screenHeight}];
250   PrintToOutputNb[text_] :=
251     (
252
```

```
253           SelectionMove[outputNb, After, Notebook];
254           NotebookWrite[outputNb, Cell[BoxData[ToBoxes[text]], "
      Output"]];
255         );
256       Return[PrintToOutputNb]
257       )
258       ]
259
260   GetModificationDate::usage="GetModificationDate[fname] returns the
      modification date of the given file.";
261   GetModificationDate[theFileName_] := FileDate[theFileName, "
      Modification"];
262
263   (*Helper function to convert Mathematica expressions to standard
      form*)
264   StandardFormExpression[expr0_] := Module[{expr=expr0}, ToString[
      expr, InputForm]];
265
266   (*Helper function to translate to Python/SymPy expressions*)
267   ToPythonSymPyExpression::usage="ToPythonSymPyExpression[expr]
      converts a Mathematica expression to a SymPy expression. This is a
       little iffy and might break if the expression includes
      Mathematica functions that haven't been given a SymPy equivalent."
      ;
268   ToPythonSymPyExpression[expr0_] := Module[{standardForm, expr=expr0
      },
269     standardForm = StandardFormExpression[expr];
270     StringReplace[standardForm, {
271       "Power[" -> "Pow(",
272       "Sqrt[" -> "sq(",
273       "[" -> "(",
274       "]" -> ")",
275       "\\" -> "",
276       "I" -> "1j",
277       "^" -> "**",
278       (*Remove special Mathematica backslashes*)
279       "/" -> "/" (*Ensure division is represented with a slash*)}]];
280
281   ToPythonSparseFunction::usage="ToPythonSparseFunction[array,
      funName] takes a 2D array (whose elements are linear combinations
      of a set of variables) and returns a string which defines a Python
       function that can be used to evaluate the given array in Python.
      In Python the name of the function is equal to funName. The given
      array may be a list of lists or a 2D SparseArray.";
282   ToPythonSparseFunction[arrayInput_, funName_] :=
283     Module[{
284       sparseArray = arrayInput,
285       rowPointers,
286       dimensions,
287       pyCode,
288       vars,
289       varList,
290       dataPyList,
291       colIndicesPyList
292       },
293     (
294       If[Head[sparseArray] === List,
295         sparseArray = SparseArray[sparseArray]
296         ];
297       (* Extract unique symbolic variables from the SparseArray *)
298       vars       = Union[Cases[Normal[sparseArray], _Symbol, Infinity
      ]];
299       varList    = StringRiffle[ToString /@ vars, ", "];
300       (* Convert data to SymPy compatible strings *)
301       dataPyList = StringRiffle[ToPythonSymPyExpression /@ Normal[
      sparseArray["NonzeroValues"]], ",\n        "];
302       colIndicesPyList = StringRiffle[
303         ToPythonSymPyExpression /@ (Flatten[Normal[sparseArray["
      ColumnIndices"]] - 1]), ", "];
304       (* Extract sparse array properties *)
305       rowPointers = Normal[sparseArray["RowPointers"]];
306       dimensions = Dimensions[sparseArray];
307       (*Create Python code string*)
308       pyCode = StringJoin[
309         "#!/usr/bin/env python3\n\n",
310         "from scipy.sparse import csr_matrix\n",
```

```
        "import numpy as np\n",
        "\n",
        "sq = np.sqrt\n",
        "\n",
        "def ", funName, "(",
        varList,
        "):\n",
        "    data = np.array([\n        ", dataPyList, "\n        ])\n",
        "    indices = np.array([",
        colIndicesPyList,
        "])\n",
        "    indptr = np.array([",
        StringRiffle[ToString /@ rowPointers, ", "], "])\n",
        "    shape = (", StringRiffle[ToString /@ dimensions, ", "],
        ")\n",
        "    return csr_matrix((data, indices, indptr), shape=shape)\n"];
      Return[pyCode];
    )
    ];

  Options[HamTeX] = {"T2" -> False};
  HamTeX::usage="HamTeX[numE] returns the LaTeX code for the semi-empirical Hamiltonian for f^numE. The option \"T2\" can be used to specify whether the T2 term should be included in the Hamiltonian for the f^12 configuration. The default is False and the option is ignored if the number of electrons is not 12.";
  HamTeX[nE_, OptionsPattern[]] := (
    tex = Which[
      MemberQ[{1, 13}, nE],
        "\\hat{H}&=\\zeta \\sum_{i=1}^{n}\\left(\\hat{s}_i \\cdot \\hat{l}_i\\right) \
    +\\sum_{i=1}^n\\sum_{k=2,4,6}\\sum_{q=-k}^{k}B_q^{(k)}\\mathcal{C}(i)_\
    q^{(k)} + \\epsilon",
      nE == 2,
        "\\hat{H}&=\\sum_{k=2,4,6}F^{(k)}\\hat{f}_k
    +\\alpha \\hat{L}^2
    +\\beta \\,\\mathcal{C}\\left(\\mathcal{G}(2)\\right)
    +\\gamma \\,\\mathcal{C}\\left(\\mathcal{SO}(7)\\right)\\\\
    &\\quad + \\zeta \\sum_{i=1}^{n}\\left(\\hat{s}_i \\cdot \\hat{l}_i\\right)
    +\\sum_{k=0,2,4}M^{(k)}\\hat{m}_k
    +\\sum_{k=2,4,6}P^{(k)}\\hat{p}_k \\\\
    &\\quad\\quad\\quad\\quad\\quad+\\sum_{i=1}^n\\sum_{k=2,4,6}\\sum_{q=-\
    k}^{k}B_q^{(k)}\\mathcal{C}(i)_q^{(k)} + \\epsilon",
      And[nE == 12, OptionValue["T2"]],
        "\\hat{H}&=\\sum_{k=2,4,6}F^{(k)}\\hat{f}_k
        +T^{(2)}\\hat{t}_2
        +\\alpha \\hat{L}^2
        +\\beta \\,\\mathcal{C}\\left(\\mathcal{G}(2)\\right)
        +\\gamma \\,\\mathcal{C}\\left(\\mathcal{SO}(7)\\right)\\\\
        &\\quad\\quad\\quad + \\zeta \\sum_{i=1}^{n}\\left(\\hat{s}_i \\cdot \\hat{l}_i\\right)
        +\\sum_{k=0,2,4}M^{(k)}\\hat{m}_k
        +\\sum_{k=2,4,6}P^{(k)}\\hat{p}_k \\\\
        &\\quad\\quad\\quad\\quad\\quad\\quad\\quad+\\sum_{i=1}^n\\sum_{k=2,4,\
        6}\\sum_{q=-k}^{k}B_q^{(k)}\\mathcal{C}(i)_q^{(k)} + \\epsilon",
      And[nE == 12, Not@OptionValue["T2"]],
        "\\hat{H}&=\\sum_{k=2,4,6}F^{(k)}\\hat{f}_k
        +\\alpha \\hat{L}^2
        +\\beta \\,\\mathcal{C}\\left(\\mathcal{G}(2)\\right)
        +\\gamma \\,\\mathcal{C}\\left(\\mathcal{SO}(7)\\right)\\\\
        &\\quad + \\zeta \\sum_{i=1}^{n}\\left(\\hat{s}_i \\cdot \\hat{l}_i\\right)
        +\\sum_{k=0,2,4}M^{(k)}\\hat{m}_k
        +\\sum_{k=2,4,6}P^{(k)}\\hat{p}_k \\\\
        &\\quad\\quad\\quad+\\sum_{i=1}^n\\sum_{k=2,4,6}\\sum_{q=-k}^{k}B_q^{(\
    k)}\\mathcal{C}(i)_q^{(k)} + \\epsilon",
      True,
```

```
375         "\\hat{H}&=\\sum_{k=2,4,6}F^{(k)}\\hat{f}_k
376         +\\sum_{k=2,3,4,6,7,8}T^{(k)}\\hat{t}_k
377         +\\alpha \\hat{L}^2
378         +\\beta \\,\\mathcal{C}\\left(\\mathcal{G}(2)\\right)
379         +\\gamma \\,\\mathcal{C}\\left(\\mathcal{SO}(7)\\right)\\\\
380         &\\quad\\quad\\quad\\quad + \\zeta \\sum_{i=1}^{n}\\left(\\
    hat{s}_i \
381         \\cdot \\hat{l}_i\\right)
382         +\\sum_{k=0,2,4}M^{(k)}\\hat{m}_k
383         +\\sum_{k=2,4,6}P^{(k)}\\hat{p}_k \\\\
384         &\\quad\\quad\\quad\\quad\\quad\\quad\\quad\\quad+\\sum_{i
    =1}^n\\sum_{\
385         k=2,4,6}\\sum_{q=-k}^{k}B_q^{(k)}\\mathcal{C}(i)_q^{(k)} + \\
    epsilon"
386       ];
387     Return[StringJoin[{"\\begin{aligned}\n", tex, "\n\\end{aligned}"
    }]]
388     )
389
390   EllipsoidBoundingBox::usage = "EllipsoidBoundingBox[A,\[Kappa]]
      gives the coordinate intervals that contain the ellipsoid
      determined by r^T.A.r==\[Kappa]^2. The matrix A must be square NxN
      , symmetric, and positive definite. The function returns a list
      with N pairs of numbers, each pair being of the form {-x_i, x_i}."
      ;
391   EllipsoidBoundingBox[Amat_,\[Kappa]_]:=Module[
392     {invAmat, stretchFactors, boundingPlanes, quad},
393       (
394       invAmat = Inverse[Amat];
395       stretchFactors = Sqrt[1/Diagonal[invAmat]];
396       boundingPlanes = DiagonalMatrix[stretchFactors].invAmat;
397       (* The solution is proportional to \[Kappa] *)
398       boundingPlanes = \[Kappa] * boundingPlanes;
399       boundingPlanes = Max /@ Transpose[boundingPlanes];
400       Return[{-#,#}& /@ boundingPlanes]
401     )
402   ];
403
404 End[];
405 EndPackage[];
```

References

[BCR99] Gary W Burdick, SM Crooks, and MF Reid. "Ambiguities in the parametrization of 4 f N- 4 f N electric-dipole transition intensities". In: *Physical Review B* 59.12 (1999). Publisher: APS, R7789.

[BG15] Cristiano Benelli and Dante Gatteschi. *Introduction to molecular magnetism: from transition metals to lanthanides*. John Wiley & Sons, 2015.

[BG34] R. F. Bacher and S. Goudsmit. "Atomic Energy Relations. I". In: *Phys. Rev.* 46.11 (Dec. 1934). Publisher: American Physical Society, pp. 948–969.

[BS57] Hans Bethe and Edwin Salpeter. *Quantum Mechanics of One- and Two-Electron Atoms*. 1957.

[BSR88] Mary T Berry, Charles Schwieters, and FS Richardson. "Optical absorption spectra, crystal-field analysis, and electric dipole intensity parameters for europium in Na3 [En (ODA) 3]-2NaClO4 · 6H2O". In: *Chemical physics* 122.1 (1988). Publisher: Elsevier, pp. 105–124.

[Bur+94] G. W. Burdick et al. "Energy-level and line-strength analysis of optical transitions between Stark levels in \mathrmNd^3+:\mathrmY_3\mathrmAl_5\mathrmO_12". In: *Phys. Rev. B* 50.22 (1994). Publisher: American Physical Society, pp. 16309–16325.

[Car+70] WT Carnall et al. "Absorption spectrum of Tm3+: LaF3". In: *The Journal of Chemical Physics* 52.8 (1970). Publisher: American Institute of Physics, pp. 4054–4059.

[Car+76] WT Carnall et al. "Energy level analysis of Pm3+: LaCl3". In: *The Journal of Chemical Physics* 64.9 (1976). Publisher: American Institute of Physics, pp. 3582–3591.

[Car+89] W. T. Carnall et al. "A systematic analysis of the spectra of the lanthanides doped into single crystal LaF3". en. In: *The Journal of Chemical Physics* 90.7 (1989), pp. 3443–3457. ISSN: 0021-9606, 1089-7690.

[Car92] William T Carnall. "A systematic analysis of the spectra of trivalent actinide chlorides in D3h site symmetry". In: *The Journal of chemical physics* 96.12 (1992). Publisher: American Institute of Physics, pp. 8713–8726.

[CCJ68] Hannah Crosswhite, HM Crosswhite, and BR Judd. "Magnetic Parameters for the Configuration f 3". In: *Physical Review* 174.1 (1968). Publisher: APS, p. 89.

[CFR68a] WT Carnall, PR Fields, and K Rajnak. "Electronic energy levels in the trivalent lanthanide aquo ions. I. Pr3+, Nd3+, Pm3+, Sm3+, Dy3+, Ho3+, Er3+, and Tm3+". In: *The Journal of chemical physics* 49.10 (1968). Publisher: American Institute of Physics, pp. 4424–4442.

[CFR68b] WT Carnall, PR Fields, and K Rajnak. "Electronic energy levels of the trivalent lanthanide aquo ions. IV. Eu3+". In: *The Journal of Chemical Physics* 49.10 (1968). Publisher: American Institute of Physics, pp. 4450–4455.

[CFR68c] WT Carnall, PR Fields, and K Rajnak. "Electronic energy levels of the trivalent lanthanide aquo ions. III. Tb3+". In: *The Journal of chemical physics* 49.10 (1968). Publisher: American Institute of Physics, pp. 4447–4449.

[CFR68d] WT Carnall, PR Fields, and K Rajnak. "Electronic energy levels of the trivalent lanthanide aquo ions. II. Gd3+". In: *The Journal of chemical physics* 49.10 (1968). Publisher: American Institute of Physics, pp. 4443–4446.

[CFR68e] WT Carnall, PR Fields, and K Rajnak. "Spectral intensities of the trivalent lanthanides and actinides in solution. II. Pm3+, Sm3+, Eu3+, Gd3+, Tb3+, Dy3+, and Ho3+". In: *The journal of chemical physics* 49.10 (1968). Publisher: American Institute of Physics, pp. 4412–4423.

[CFW65] W To Carnall, PR Fields, and BG Wybourne. "Spectral intensities of the trivalent lanthanides and actinides in solution. I. Pr3+, Nd3+, Er3+, Tm3+, and Yb3+". In: *The Journal of Chemical Physics* 42.11 (1965). Publisher: American Institute of Physics, pp. 3797–3806.

[Che+08] Xueyuan Chen et al. “A few mistakes in widely used data files for fn configurations calculations”. In: *Journal of luminescence* 128.3 (2008). Publisher: Elsevier, pp. 421–427.

[Che+16] Jun Cheng et al. “Crystal-field analyses for trivalent lanthanide ions in LiYF4”. In: *Journal of Rare Earths* 34.10 (2016). Publisher: Elsevier, pp. 1048–1052.

[Cow81] Robert Duane Cowan. *The theory of atomic structure and spectra*. en. Los Alamos series in basic and applied sciences 3. Berkeley: University of California Press, 1981. ISBN: 978-0-520-03821-9.

[Cro+76] HM Crosswhite et al. “The spectrum of Nd3+: LaCl3”. In: *The Journal of Chemical Physics* 64.5 (1976). Publisher: American Institute of Physics, pp. 1981–1985.

[Cro+77] HM Crosswhite et al. “Parametric energy level analysis of Ho3+: LaCl3”. In: *The Journal of Chemical Physics* 67.7 (1977). Publisher: American Institute of Physics, pp. 3002–3010.

[Cro71] HM Crosswhite. “Effective electrostatic operators for two inequivalent electrons”. In: *Physical Review A* 4.2 (1971). Publisher: APS, p. 485.

[CW63] JG Conway and BG Wybourne. “Low-lying energy levels of lanthanide atoms and intermediate coupling”. In: *Physical Review* 130.6 (1963). Publisher: APS, p. 2325.

[DC63] G. H. Dieke and H. M. Crosswhite. “The Spectra of the Doubly and Triply Ionized Rare Earths”. en. In: *Applied Optics* 2.7 (July 1963), p. 675. ISSN: 0003-6935, 1539-4522.

[Die68] G. H. Dieke. *Spectra and Energy Levels of Rare Earth Ions in Crystals*. Ed. by Hannah Crosswhite and H. M. Crosswhite. 1968.

[DR06] Chang-Kui Duan and Michael F Reid. “Dependence of the spontaneous emission rates of emitters on the refractive index of the surrounding media”. In: *Journal of alloys and compounds* 418.1-2 (2006). Publisher: Elsevier, pp. 213–216.

[DZ12] Christopher M. Dodson and Rashid Zia. “Magnetic dipole and electric quadrupole transitions in the trivalent lanthanide series: Calculated emission rates and oscillator strengths”. en. In: *Physical Review B* 86.12 (Sept. 2012), p. 125102. ISSN: 1098-0121, 1550-235X.

[GW+91] C Görller-Walrand et al. “Magnetic dipole transitions as standards for Judd–Ofelt parametrization in lanthanide spectra”. In: *The Journal of chemical physics* 95.5 (1991). Publisher: American Institute of Physics, pp. 3099–3106.

[JC84] BR Judd and Hannah Crosswhite. “Orthogonalized operators for the f shell”. In: *JOSA B* 1.2 (1984). Publisher: Optica Publishing Group, pp. 255–260.

[JCC68] BR Judd, HM Crosswhite, and Hannah Crosswhite. “Intra-atomic magnetic interactions for f electrons”. In: *Physical Review* 169.1 (1968). Publisher: APS, p. 130.

[JL93] BR Judd and GMS Lister. “Symmetries of the f shell”. In: *Journal of alloys and compounds* 193.1-2 (1993). Publisher: Elsevier, pp. 155–159.

[JS84] BR Judd and MA Suskin. “Complete set of orthogonal scalar operators for the configuration f^3”. In: *JOSA B* 1.2 (1984). Publisher: Optica Publishing Group, pp. 261–265.

[Jud05] Brian R Judd. “Interaction with William Carnall”. In: *Journal of Solid State Chemistry* 178.2 (2005). Publisher: Elsevier, pp. 408–411.

[Jud62] B. R. Judd. “Optical Absorption Intensities of Rare-Earth Ions”. en. In: *Physical Review* 127.3 (Aug. 1962), pp. 750–761. ISSN: 0031-899X.

[Jud63a] B R Judd. “Configuration Interaction in Rare Earth Ions”. en. In: *Proceedings of the Physical Society* 82.6 (Dec. 1963), pp. 874–881. ISSN: 0370-1328.

[Jud63b] Brian R. Judd. *Operator techniques in atomic spectroscopy*. en. Princeton landmarks in mathematics and physics. Princeton, N.J: Princeton University Press, 1963. ISBN: 978-0-691-05901-3.

[Jud66] BR Judd. “Three-particle operators for equivalent electrons”. In: *Physical Review* 141.1 (1966). Publisher: APS, p. 4.

[Jud67] Brian R Judd. *Second quantization and atomic spectroscopy*. 1967.

[Jud82] BR Judd. “Parametric fits in the atomic d shell”. In: *Journal of Physics B: Atomic and Molecular Physics* 15.10 (1982). Publisher: IOP Publishing, p. 1457.

[Jud83] BR Judd. “Operator averages and orthogonalities”. In: *Group Theoretical Methods in Physics: Proceedings of the XIIth International Colloquium Held at the International Centre for Theoretical Physics, Trieste, Italy, September 5–11, 1983*. Springer, 1983, pp. 340–342.

[Jud85] BR Judd. “Complex atomic spectra”. In: *Reports on Progress in Physics* 48.7 (1985). Publisher: IOP Publishing, p. 907.

[Jud86] BR Judd. “Classification of Operators in Atomic Spectroscopy by Lie Groups”. In: *Symmetries in Science II*. Springer, 1986, pp. 265–269.

[Jud88] BR Judd. “Atomic theory and optical spectroscopy”. In: *Handbook on the physics and chemistry of rare earths* 11 (1988). Publisher: Elsevier, pp. 81–195.

[Jud89] BR Judd. “Developments in the Theory of Complex Spectra”. In: *Physica Scripta* 1989.T26 (1989). Publisher: IOP Publishing, p. 29.

[Jud96] Brian R Judd. “Group Theory for atomic shells”. In: *Springer Handbook of Atomic, Molecular, and Optical Physics*. Springer, 1996, pp. 71–80.

[Lea82] Richard P. Leavitt. “On the role of certain rotational invariants in crystal-field theory”. en. In: *The Journal of Chemical Physics* 77.4 (Aug. 1982), pp. 1661–1663. ISSN: 0021-9606, 1089-7690.

[Lea87] RC Leavitt. “A complete set of f-electron scalar operators”. In: *Journal of Physics A: Mathematical and General* 20.11 (1987). Publisher: IOP Publishing, p. 3171.

[Lin74] Ingvar Lindgren. “The Rayleigh-Schrodinger perturbation and the linked-diagram theorem for a multi-configurational model space”. In: *Journal of Physics B: Atomic and Molecular Physics* 7.18 (1974). Publisher: IOP Publishing, p. 2441.

[LM80] RP Leavitt and CA Morrison. “Crystal-field analysis of triply ionized rare earth ions in lanthanum trifluoride. II. Intensity calculations”. In: *The Journal of Chemical Physics* 73.2 (1980). Publisher: American Institute of Physics, pp. 749–757.

[MJR89a] P Stanley May, CK Jayasankar, and FS Richardson. “Parametric analysis of ff transition intensities in trigonal Na3 [Nd (oxydiacetate) 3] · 2NaClO4 · 6H2O”. In: *Chemical physics* 138.1 (1989). Publisher: Elsevier, pp. 139–156.

[MJR89b] P Stanley May, CK Jayasankar, and FS Richarson. “Crystals field energy levels and transition line strengths of neodymium in trigonal Na3 [Nd (oxydiacetate) 3] · 2NaClO4 · 6H2O”. In: *Chemical physics* 138.1 (1989). Publisher: Elsevier, pp. 123–138.

[MKW77a] Clyde A Morrison, Nick Karayianis, and Donald E Wortman. *Rare-Earth Ion-Host Lattice Interactions. 4. Predicting Spectra and Intensities of Lanthanides in Crystals.* Tech. rep. HARRY DIAMOND LABS ADELPHI MD, 1977.

[MKW77b] Clyde A Morrison, Nick Karayianis, and Donald E Wortman. *Theoretical Free-Ion Energies, Derivatives and Reduced Matrix Elements I. Pr (3+), Tm (3+), Nd (3+), and Er (3+).* Tech. rep. HARRY DIAMOND LABS ADELPHI MD, 1977.

[ML79] CA Morrison and RP Leavitt. “Crystal-field analysis of triply ionized rare earth ions in lanthanum trifluoride”. In: *The Journal of Chemical Physics* 71.6 (1979). Publisher: American Institute of Physics, pp. 2366–2374.

[ML82] Clyde A Morrison and Richard P Leavitt. “Spectroscopic properties of triply ionized”. In: *Handbook on the physics and chemistry of rare earths* 5 (1982). Publisher: Elsevier, pp. 461–692.

[Mor+83] Clyde A Morrison et al. “Optical spectra, energy levels, and crystal-field analysis of tripositive rare-earth ions in Y2O3. III. Intensities and g values for C2 sites”. In: *The Journal of chemical physics* 79.10 (1983). Publisher: American Institute of Physics, pp. 4758–4763.

[Mor80] Clyde Morrison. “Host dependence of the rare-earth ion energy separation 4f N–4f N−1 nl”. In: *The Journal of Chemical Physics* (1980).

[MR90] DM Moran and FS Richardson. “Parametric analysis of f-f transition intensities in trigonal Na 3 [Ho (C 4 H 4 O 5) 3] 2 N a C l O 4 6 H 2 O”. In: *Physical Review B* 42.6 (1990). Publisher: APS, p. 3331.

[MR92] Diane M Moran and FS Richardson. “Measurement and analysis of circular dichroism in the 4f-4f transitions of holmium (3+) in sodium tris (oxydiacetato) holmate (3-) bis (sodium perchlorate) hexahydrate”. In: *Inorganic Chemistry* 31.5 (1992). Publisher: ACS Publications, pp. 813–818.

[MRR87a] P Stanley May, Michael F Reid, and FS Richardson. “Circular dichroism spectra and electronic rotatory strengths of the samarium 4 f→ 4 f transitions in Na3 [Sm (oxydiacetate) 3] · 2NaClO4 · 6H2O”. In: *Molecular Physics* 62.2 (1987). Publisher: Taylor & Francis, pp. 341–364.

[MRR87b] P Stanley May, Michael F Reid, and FS Richardson. “Electric dipole intensity parameters for the samarium 4 f→ 4 f transitions in Na3 [Sm (oxydiacetate) 3] · 2NaClO4 · 6H2O”. In: *Molecular Physics* 61.6 (1987). Publisher: Taylor & Francis, pp. 1471–1485.

[MRR87c] P Stanley May, Michael F Reid, and FS Richardson. “Optical spectra, energy levels and crystal-field analysis of Sm3+ in Na3 [Sm (oxydiacetate) 3] · 2NaClO4 · 6H2O”. In: *Molecular Physics* 61.6 (1987). Publisher: Taylor & Francis, pp. 1455–1470.

[MT87] Clyde Morrison and Gregory Turner. *Analysis of the Optical Spectra of Triply Ionized Transition Metal Ions in Yttrium Aluminum Garnet*. Tech. rep. 1987.

[MW94] CA Morrison and DE Wortman. *Energy Levels, Transition Probabilities, and Branching Ratios for Rare-Earth Ions in Transparent Solids*. SPIE Optical Engineering Press, 1994.

[MWK76] CA Morrison, DE Wortman, and N Karayianis. “Crystal-field parameters for triply-ionized lanthanides in yttrium aluminium garnet”. In: *Journal of Physics C: Solid State Physics* 9.8 (1976). Publisher: IOP Publishing, p. L191.

[NB75] DJ Newman and G Balasubramanian. “Parametrization of rare-earth ion transition intensities”. In: *Journal of Physics C: Solid State Physics* 8.1 (1975). Publisher: IOP Publishing, p. 37.

[New82] DJ Newman. “Operator orthogonality and parameter uncertainty”. In: *Physics Letters A* 92.4 (1982). Publisher: Elsevier, pp. 167–169.

[NK63] C. W. Nielson and George F Koster. *Spectroscopic Coefficients for the pn, dn, and fn configurations*. 1963.

[NN00] Douglas John Newman and Betty Ng. *Crystal field handbook*. Vol. 2007. Cambridge University Press Cambridge, 2000.

[Ofe62] GS Ofelt. “Intensities of crystal spectra of rare-earth ions”. In: *The journal of chemical physics* 37.3 (1962). Publisher: American Institute of Physics, pp. 511–520.

[PDC67] AH Piksis, GH Dieke, and HM Crosswhite. “Energy levels and crystal field of LaCl3: Gd3+”. In: *The Journal of Chemical Physics* 47.12 (1967). Publisher: American Institute of Physics, pp. 5083–5089.

[Rac42a] Giulio Racah. “Theory of Complex Spectra. I”. en. In: *Physical Review* 61.3-4 (Feb. 1942), pp. 186–197. ISSN: 0031-899X.

[Rac42b] Giulio Racah. “Theory of Complex Spectra. II”. en. In: *Physical Review* 62.9-10 (Nov. 1942), pp. 438–462. ISSN: 0031-899X.

[Rac43] Giulio Racah. “Theory of Complex Spectra. III”. en. In: *Physical Review* 63.9-10 (May 1943), pp. 367–382. ISSN: 0031-899X.

[Rac49] Giulio Racah. “Theory of Complex Spectra. IV”. en. In: *Physical Review* 76.9 (Nov. 1949), pp. 1352–1365. ISSN: 0031-899X.

[Raj65] K Rajnak. “Configuration Interaction in the 4f 3 Configuration of Pr iii”. In: *JOSA* 55.2 (1965). Publisher: Optica Publishing Group, pp. 126–132.

[Rei81] Michael F Reid. “Applications of Group Theory in Solid State Physics”. PhD thesis. University of Canterbury, 1981.

[Rei87] Michael F Reid. “Superposition-model analysis of intensity parameters for Eu3+ luminescence”. In: *The Journal of chemical physics* 87.11 (1987). Publisher: American Institute of Physics, pp. 6388–6392.

[Rei92] Michael F Reid. “Recent extensions to crystal-field and transition-intensity models”. In: *Journal of alloys and compounds* 180.1-2 (1992). Publisher: Elsevier, pp. 93–103.

[RR83a] Michael F Reid and Frederick S Richardson. “Anisotropic ligand polarizability contributions to lanthanide 4f→ 4f intensity parameters”. In: *Chemical Physics Letters* 95.6 (1983). Publisher: Elsevier, pp. 501–506.

[RR83b] Michael F Reid and FS Richardson. “Electric dipole intensity parameters for lanthanide 4f→ 4f transitions”. In: *The Journal of chemical physics* 79.12 (1983). Publisher: AIP Publishing, pp. 5735–5742.

[RR84] Michael F Reid and FS Richardson. “Lanthanide 4f -> 4f electric dipole intensity theory”. In: *The Journal of Physical Chemistry* 88.16 (1984). Publisher: ACS Publications, pp. 3579–3586.

[Rud07] Zenonas Rudzikas. *Theoretical atomic spectroscopy*. 2007.

[RW63] K Rajnak and BG Wybourne. “Configuration interaction effects in l^N configurations”. In: *Physical Review* 132.1 (1963). Publisher: APS, p. 280.

[RW64a] K Rajnak and BG Wybourne. “Configuration interaction in crystal field theory”. In: *The Journal of Chemical Physics* 41.2 (1964). Publisher: American Institute of Physics, pp. 565–569.

[RW64b] K Rajnak and BG Wybourne. “Electrostatically correlated spin-orbit interactions in l n-type configurations”. In: *Physical Review* 134.3A (1964). Publisher: APS, A596.

[Sla29] J. C. Slater. “The Theory of Complex Spectra”. en. In: *Physical Review* 34.10 (Nov. 1929), pp. 1293–1322. ISSN: 0031-899X.

[TLJ99] Anne Thorne, Ulf Litzén, and Sveneric Johansson. *Spectrophysics: principles and applications*. Springer Science & Business Media, 1999.

[Tre51] RE Trees. “Spin-spin interaction”. In: *Physical Review* 82.5 (1951). Publisher: APS, p. 683.

[Tre52] R. E. Trees. “The L (L + 1) Correction to the Slater Formulas for the Energy Levels”. en. In: *Physical Review* 85.2 (Jan. 1952), pp. 382–382. ISSN: 0031-899X.

[Tre58] Richard E. Trees. “Comparison of First, Second, and Third Approximations in Bacher and Goudsmit’s Theory of Atomic Spectra”. In: *J. Opt. Soc. Am.* 48.5 (May 1958). Publisher: Optica Publishing Group, pp. 293–300.

[Vel00] Dobromir Velkov. “Multi-electron coefficients of fractional parentage for the p, d, and f shells”. PhD thesis. John Hopkins University, 2000.

[WS07] Brian Wybourne and Lidia Smentek. *Optical Spectroscopy of Lanthanides*. 2007.

[Wyb63] BG Wybourne. “Electrostatic Interactions in Complex Electron Configurations”. In: *Journal of Mathematical Physics* 4.3 (1963). Publisher: American Institute of Physics, pp. 354–356.

[Wyb64a] BG Wybourne. “Low-Lying Energy Levels of Trivalent Curium”. In: *The Journal of Chemical Physics* 40.5 (1964). Publisher: American Institute of Physics, pp. 1456–1457.

[Wyb64b] BG Wybourne. “Orbit—Orbit Interactions and the“Linear”Theory of Configuration Interaction”. In: *The Journal of Chemical Physics* 40.5 (1964). Publisher: American Institute of Physics, pp. 1457–1458.

[Wyb65] Brian G Wybourne. *Spectroscopic Properties of Rare Earths*. 1965.

[Wyb70] Brian G Wybourne. *Symmetry principles and atomic spectroscopy*. 1970.

[Wol24a] Wolfram Research. *SixJSymbol*. 2024.

[Wol24b] Wolfram Research. *ThreeJSymbol*. 2024.

Index