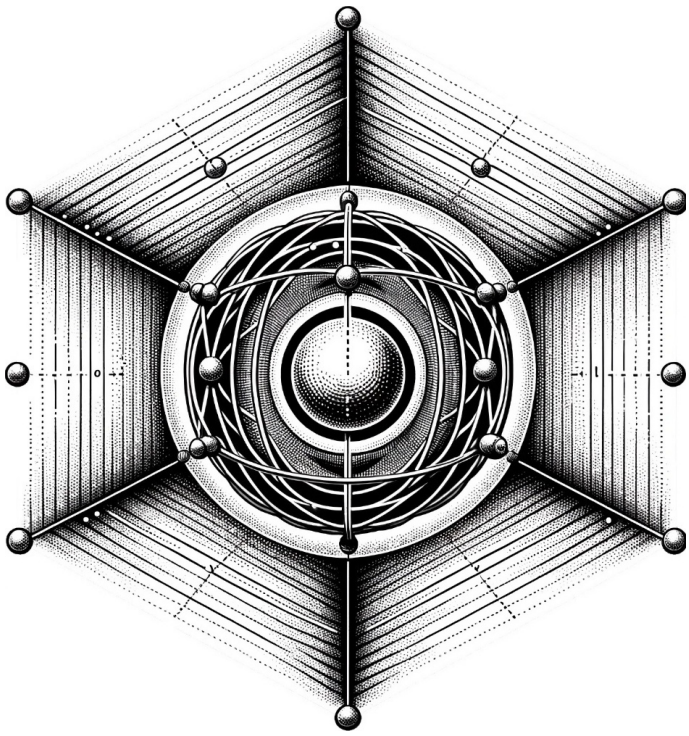

qlanth

version $|\alpha\rangle^{(7)}$

Juan David Lizarazo Ferro
& Christopher Dodson

Under the advisory of Dr. Rashid Zia

Contents

qlanth is a tool that can be used to estimate the electronic structure of lanthanide ions in crystals. For this purpose it uses a single configuration description and a corresponding effective Hamiltonian. This Hamiltonian aims to describe the observed properties of ions embedded in solids in a picture that imagines them as free-ions but modified by the influence of the lattice in which they find themselves in.

This picture is one that developed and mostly matured in the second half of the last century by the efforts of Giulio Racah, Brian Judd, Hannah Crosswhite, Robert Cowan, Michael Reid, Bill Carnall, Clyde Morrison, Richard Leavitt, Brian Wybourne, and Katherine Rajnak 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.

qlanth also includes data that might be of use to those interested in the single-configuration description of lanthanide ions, separate to their specific use in this code. 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.

The included Mathematica notebook `qlanth.nb` has examples of the capabilities that this tool offers, and the `/examples` folder includes a series of notebooks 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 modified and rewritten by David Lizarazo. It has also benefited from conversations with Tharnier Puel at the University of Iowa.

This document has 12 sections. **Section 1** explains the details of the basis in which the Hamiltonian is evaluated. **Section 2** provides a brief explanation of the coefficients of fractional parentage. **Section 3** explains how the Hamiltonian is put together by first having calculated "JJ' blocks". **Section 4** is dedicated to a theoretical exposition of the effective Hamiltonian with subsections for each of the terms that it contains. **Section 5** is about the calculation of magnetic dipole transitions. **Sections 6 and 8** list additional data included in **qlanth**. **Section 7** has additional information about data fitting. **Section 9** has a brief comment on units. **Sections 9 and 10** include a summary of notation and definitions use throughout this document. Finally, **section 12** contains a printout of the code included in **qlanth**.

1 The $|LSJM_J\rangle$ Basis States

The basis used in **qlanth** is the $|LSJM_J\rangle$ basis. As such the basis vectors are common eigenvectors of the operators $\hat{L}^2$, $\hat{S}^2$, $\hat{J}^2$, and $\hat{J}_z$. The LS terms allowed in each configuration $\underline{f}^n$ are obtained from tables that originate from the original work by Nielson and Koster [NK63]. In **qlanth** these are parsed from the file `B1F_ALL.TXT` which is part of the doctoral research of Dobromir Velkov [Vel00] in which he recomputed coefficients of fractional parentage under the advisory of Brian Judd.

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 path 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 specific role given to them, except that of discriminating between different degenerate terms.

The following are all the LS terms in the $\underline{f}^n$ configurations. In the notation used the superscript index before the letter notes the spin multiplicity $2S + 1$, the roman letter indicating the value

of L in spectroscopic notation ($S\rightarrow1, P\rightarrow2, D\rightarrow3, F\rightarrow4, G\rightarrow5, H\rightarrow6, I\rightarrow7, K\rightarrow8, L\rightarrow9, M\rightarrow10, N\rightarrow11, O\rightarrow12, Q\rightarrow3, R\rightarrow14, T\rightarrow15, U\rightarrow16, V\rightarrow17$), and the final integer (if present) is the label that discriminates between several degenerate LS. This index we frequently label in the equations contained in this document with the greek letter α.

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

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

$\underline{f}^2$ (7 LS terms)
^{3}P, ^{3}F, ^{3}H, ^{1}S, ^{1}D, ^{1}G, ^{1}I

$\underline{f}^3$ (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

$\underline{f}^4$ (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

$\underline{f}^5$ (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

$\underline{f}^6$ (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

$\underline{\mathrm{f}}^{7}$
(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

$\underline{\mathrm{f}}^{8}$
(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

$\underline{\mathrm{f}}^{9}$
(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

$\underline{\mathrm{f}}^{10}$
(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

$\underline{\mathrm{f}}^{11}$
(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 basis vector 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 the one 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 kets)

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

$(J=1)$
(3 kets)

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

$(J=2)$
(15 kets)

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

($J = 3$) (7 kets)
$\vert^3\mathrm{F}, 3, -3\rangle$, $\vert^3\mathrm{F}, 3, -2\rangle$, $\vert^3\mathrm{F}, 3, -1\rangle$, $\vert^3\mathrm{F}, 3, 0\rangle$, $\vert^3\mathrm{F}, 3, 1\rangle$, $\vert^3\mathrm{F}, 3, 2\rangle$, $\vert^3\mathrm{F}, 3, 3\rangle$

($J = 4$) (27 kets)
$\vert^3\mathrm{F}, 4, -4\rangle$, $\vert^3\mathrm{F}, 4, -3\rangle$, $\vert^3\mathrm{F}, 4, -2\rangle$, $\vert^3\mathrm{F}, 4, -1\rangle$, $\vert^3\mathrm{F}, 4, 0\rangle$, $\vert^3\mathrm{F}, 4, 1\rangle$, $\vert^3\mathrm{F}, 4, 2\rangle$, $\vert^3\mathrm{F}, 4, 3\rangle$, $\vert^3\mathrm{F}, 4, 4\rangle$, $\vert^3\mathrm{H}, 4, -4\rangle$, $\vert^3\mathrm{H}, 4, -3\rangle$, $\vert^3\mathrm{H}, 4, -2\rangle$, $\vert^3\mathrm{H}, 4, -1\rangle$, $\vert^3\mathrm{H}, 4, 0\rangle$, $\vert^3\mathrm{H}, 4, 1\rangle$, $\vert^3\mathrm{H}, 4, 2\rangle$, $\vert^3\mathrm{H}, 4, 3\rangle$, $\vert^3\mathrm{H}, 4, 4\rangle$, $\vert^1\mathrm{G}, 4, -4\rangle$, $\vert^1\mathrm{G}, 4, -3\rangle$, $\vert^1\mathrm{G}, 4, -2\rangle$, $\vert^1\mathrm{G}, 4, -1\rangle$, $\vert^1\mathrm{G}, 4, 0\rangle$, $\vert^1\mathrm{G}, 4, 1\rangle$, $\vert^1\mathrm{G}, 4, 2\rangle$, $\vert^1\mathrm{G}, 4, 3\rangle$, $\vert^1\mathrm{G}, 4, 4\rangle$

($J = 5$) (11 kets)
$\vert^3\mathrm{H}, 5, -5\rangle$, $\vert^3\mathrm{H}, 5, -4\rangle$, $\vert^3\mathrm{H}, 5, -3\rangle$, $\vert^3\mathrm{H}, 5, -2\rangle$, $\vert^3\mathrm{H}, 5, -1\rangle$, $\vert^3\mathrm{H}, 5, 0\rangle$, $\vert^3\mathrm{H}, 5, 1\rangle$, $\vert^3\mathrm{H}, 5, 2\rangle$, $\vert^3\mathrm{H}, 5, 3\rangle$, $\vert^3\mathrm{H}, 5, 4\rangle$, $\vert^3\mathrm{H}, 5, 5\rangle$

($J = 6$) (26 kets)
$\vert^3\mathrm{H}, 6, -6\rangle$, $\vert^3\mathrm{H}, 6, -5\rangle$, $\vert^3\mathrm{H}, 6, -4\rangle$, $\vert^3\mathrm{H}, 6, -3\rangle$, $\vert^3\mathrm{H}, 6, -2\rangle$, $\vert^3\mathrm{H}, 6, -1\rangle$, $\vert^3\mathrm{H}, 6, 0\rangle$, $\vert^3\mathrm{H}, 6, 1\rangle$, $\vert^3\mathrm{H}, 6, 2\rangle$, $\vert^3\mathrm{H}, 6, 3\rangle$, $\vert^3\mathrm{H}, 6, 4\rangle$, $\vert^3\mathrm{H}, 6, 5\rangle$, $\vert^3\mathrm{H}, 6, 6\rangle$, $\vert^1\mathrm{I}, 6, -6\rangle$, $\vert^1\mathrm{I}, 6, -5\rangle$, $\vert^1\mathrm{I}, 6, -4\rangle$, $\vert^1\mathrm{I}, 6, -3\rangle$, $\vert^1\mathrm{I}, 6, -2\rangle$, $\vert^1\mathrm{I}, 6, -1\rangle$, $\vert^1\mathrm{I}, 6, 0\rangle$, $\vert^1\mathrm{I}, 6, 1\rangle$, $\vert^1\mathrm{I}, 6, 2\rangle$, $\vert^1\mathrm{I}, 6, 3\rangle$, $\vert^1\mathrm{I}, 6, 4\rangle$, $\vert^1\mathrm{I}, 6, 5\rangle$, $\vert^1\mathrm{I}, 6, 6\rangle$

The order above is exemplar of the ordering in the bases. Notice how the basis vectors are sorted in order of increasing J, so that for instance not all of the basis kets 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 $\underline{f}^n$ is provided by `BasisLSJMJ`.

```
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]
  )
];
```

1.1 The Intermediate Coupling $|LSJ\rangle$ 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. In this case, often called *intermediate coupling*, the following are the different $^{2S+1}\mathrm{L}_J$ levels that span the eigenvectors that result from diagonalizing the intermediate coupling Hamiltonian.

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

$\underline{f}^5$ (198 LSJ levels)

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

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

$^7F_0$, $^5D_0$, $^5D_0$, $^5D_0$, $^5D_0$, $^3P_0$, $^3P_0$, $^3P_0$, $^3P_0$, $^3P_0$, $^3P_0$, $^1S_0$, $^1S_0$, $^1S_0$, $^1S_0$, $^7F_1$, $^5P_1$, $^5D_1$, $^5D_1$, $^5D_1$, $^5F_1$, $^5F_1$, $^3P_1$, $^3P_1$, $^3P_1$, $^3P_1$, $^3P_1$, $^3P_1$, $^3D_1$, $^3D_1$, $^3D_1$, $^3D_1$, $^3D_1$, $^1P_1$, $^7F_2$, $^5S_2$, $^5P_2$, $^5D_2$, $^5D_2$, $^5D_2$, $^5F_2$, $^5F_2$, $^5G_2$, $^5G_2$, $^5G_2$, $^3P_2$, $^3P_2$, $^3P_2$, $^3P_2$, $^3P_2$, $^3P_2$, $^3D_2$, $^3D_2$, $^3D_2$, $^3D_2$, $^3D_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^7F_3$, $^5P_3$, $^5D_3$, $^5D_3$, $^5D_3$, $^5F_3$, $^5F_3$, $^5G_3$, $^5G_3$, $^5G_3$, $^5H_3$, $^5H_3$, $^3D_3$, $^3D_3$, $^3D_3$, $^3D_3$, $^3D_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^1F_3$, $^1F_3$, $^1F_3$, $^1F_3$, $^7F_4$, $^5D_4$, $^5D_4$, $^5D_4$, $^5F_4$, $^5F_4$, $^5G_4$, $^5G_4$, $^5G_4$, $^5H_4$, $^5H_4$, $^5I_4$, $^5I_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^7F_5$, $^5F_5$, $^5F_5$, $^5F_5$, $^5G_5$, $^5G_5$, $^5G_5$, $^5H_5$, $^5H_5$, $^5I_5$, $^5I_5$, $^5K_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3I_5$, $^3I_5$, $^3I_5$, $^3I_5$, $^3I_5$, $^3I_5$, $^1H_5$, $^1H_5$, $^1H_5$, $^1H_5$, $^7F_6$, $^5G_6$, $^5G_6$, $^5G_6$, $^5H_6$, $^5H_6$, $^5I_6$, $^5I_6$, $^5K_6$, $^5L_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3I_6$, $^3I_6$, $^3I_6$, $^3I_6$, $^3I_6$, $^3I_6$, $^3K_6$, $^3K_6$, $^3K_6$, $^3K_6$, $^3K_6$, $^3K_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^5H_7$, $^5H_7$, $^5I_7$, $^5I_7$, $^5K_7$, $^5L_7$, $^3I_7$, $^3I_7$, $^3I_7$, $^3I_7$, $^3I_7$, $^3I_7$, $^3K_7$, $^3K_7$, $^3K_7$, $^3K_7$, $^3K_7$, $^3K_7$, $^3L_7$, $^3L_7$, $^3L_7$, $^1K_7$, $^1K_7$, $^1K_7$, $^5I_8$, $^5I_8$, $^5K_8$, $^5L_8$, $^3K_8$, $^3K_8$, $^3K_8$, $^3K_8$, $^3K_8$, $^3K_8$, $^3K_8$, $^3L_8$, $^3L_8$, $^3L_8$, $^3M_8$, $^3M_8$, $^3M_8$, $^1L_8$, $^1L_8$, $^1L_8$, $^1L_8$, $^5K_9$, $^5L_9$, $^3L_9$, $^3L_9$, $^3L_9$, $^3M_9$, $^3M_9$, $^3M_9$, $^3N_9$, $^1M_9$, $^1M_9$, $^5L_{10}$, $^3M_{10}$, $^3M_{10}$, $^3M_{10}$, $^3N_{10}$, $^3O_{10}$, $^1N_{10}$, $^1N_{10}$, $^3N_{11}$, $^3O_{11}$, $^3O_{12}$, $^1Q_{12}$

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

$^6D_{1/2}$, $^6F_{1/2}$, $^4P_{1/2}$, $^4P_{1/2}$, $^4D_{1/2}$, $^4D_{1/2}$, $^4D_{1/2}$, $^4D_{1/2}$, $^4D_{1/2}$, $^4D_{1/2}$, $^2S_{1/2}$, $^2S_{1/2}$, $^2P_{1/2}$, $^2P_{1/2}$, $^2P_{1/2}$, $^2P_{1/2}$, $^2P_{1/2}$, $^2P_{1/2}$, $^6P_{3/2}$, $^6D_{3/2}$, $^6F_{3/2}$, $^6G_{3/2}$, $^4S_{3/2}$, $^4S_{3/2}$, $^4P_{3/2}$, $^4P_{3/2}$, $^4D_{3/2}$, $^4D_{3/2}$, $^4D_{3/2}$, $^4D_{3/2}$, $^4D_{3/2}$, $^4D_{3/2}$, $^4F_{3/2}$, $^4F_{3/2}$, $^4F_{3/2}$, $^4F_{3/2}$, $^4F_{3/2}$, $^2P_{3/2}$, $^2P_{3/2}$, $^2P_{3/2}$, $^2P_{3/2}$, $^2P_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^2D_{3/2}$, $^6P_{5/2}$, $^6D_{5/2}$, $^6F_{5/2}$, $^6G_{5/2}$, $^6H_{5/2}$, $^4P_{5/2}$, $^4P_{5/2}$, $^4D_{5/2}$, $^4D_{5/2}$, $^4D_{5/2}$, $^4D_{5/2}$, $^4D_{5/2}$, $^4D_{5/2}$, $^4F_{5/2}$, $^4F_{5/2}$, $^4F_{5/2}$, $^4F_{5/2}$, $^4F_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^4G_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2D_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^2F_{5/2}$, $^8S_{7/2}$, $^6P_{7/2}$, $^6D_{7/2}$, $^6F_{7/2}$, $^6G_{7/2}$,

$^{6}H_{7/2}$, $^{6}I_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{6}D_{9/2}$, $^{6}F_{9/2}$, $^{6}G_{9/2}$, $^{6}H_{9/2}$, $^{6}I_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}G_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}H_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{4}I_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}G_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{2}H_{9/2}$, $^{6}F_{11/2}$, $^{6}G_{11/2}$, $^{6}H_{11/2}$, $^{6}I_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}G_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}H_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}I_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{4}K_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}H_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{2}I_{11/2}$, $^{6}G_{13/2}$, $^{6}H_{13/2}$, $^{6}I_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}H_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}I_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}K_{13/2}$, $^{4}L_{13/2}$, $^{4}L_{13/2}$, $^{4}L_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}I_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{2}K_{13/2}$, $^{6}H_{15/2}$, $^{6}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}I_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}K_{15/2}$, $^{4}L_{15/2}$, $^{4}L_{15/2}$, $^{4}L_{15/2}$, $^{4}M_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}K_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{2}L_{15/2}$, $^{6}I_{17/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}K_{17/2}$, $^{4}L_{17/2}$, $^{4}L_{17/2}$, $^{4}L_{17/2}$, $^{4}M_{17/2}$, $^{4}N_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}L_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{2}M_{17/2}$, $^{4}L_{19/2}$, $^{4}L_{19/2}$, $^{4}L_{19/2}$, $^{4}M_{19/2}$, $^{4}N_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}M_{19/2}$, $^{2}N_{19/2}$, $^{2}N_{19/2}$, $^{4}M_{21/2}$, $^{4}N_{21/2}$, $^{2}N_{21/2}$, $^{2}N_{21/2}$, $^{2}O_{21/2}$, $^{4}N_{23/2}$, $^{2}O_{23/2}$, $^{2}Q_{23/2}$, $^{2}Q_{25/2}$

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

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

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

$^{6}F_{1/2}$, $^{4}P_{1/2}$, $^{4}P_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{4}D_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{2}P_{1/2}$, $^{6}P_{3/2}$, $^{6}F_{3/2}$, $^{4}S_{3/2}$, $^{4}P_{3/2}$, $^{4}P_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}D_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{4}F_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}P_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{2}D_{3/2}$, $^{6}P_{5/2}$, $^{6}F_{5/2}$, $^{6}H_{5/2}$, $^{4}P_{5/2}$, $^{4}P_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}D_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}F_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{4}G_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}D_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{2}F_{5/2}$, $^{6}P_{7/2}$, $^{6}F_{7/2}$, $^{6}H_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}D_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}F_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}G_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{4}H_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}F_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{2}G_{7/2}$, $^{6}F_{9/2}$, $^{6}H_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}F_{9/2}$, $^{4}G_{9/2}$,

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

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

$^5D_0$, $^3P_0$, $^3P_0$, $^3P_0$, $^1S_0$, $^1S_0$, $^5D_1$, $^5F_1$, $^3P_1$, $^3P_1$, $^3P_1$, $^3D_1$, $^3D_1$, $^5S_2$, $^5D_2$, $^5F_2$, $^5G_2$, $^3P_2$, $^3P_2$, $^3P_2$, $^3D_2$, $^3D_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^3F_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^1D_2$, $^5D_3$, $^5F_3$, $^5G_3$, $^3D_3$, $^3D_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3F_3$, $^3G_3$, $^3G_3$, $^3G_3$, $^1F_3$, $^5D_4$, $^5F_4$, $^5G_4$, $^5I_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3F_4$, $^3G_4$, $^3G_4$, $^3G_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^3H_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^1G_4$, $^5F_5$, $^5G_5$, $^5I_5$, $^3G_5$, $^3G_5$, $^3G_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3H_5$, $^3I_5$, $^3I_5$, $^1H_5$, $^1H_5$, $^5G_6$, $^5I_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3H_6$, $^3I_6$, $^3I_6$, $^3K_6$, $^3K_6$, $^1I_6$, $^1I_6$, $^1I_6$, $^5I_7$, $^3I_7$, $^3I_7$, $^3K_7$, $^3K_7$, $^3L_7$, $^1K_7$, $^5I_8$, $^3K_8$, $^3K_8$, $^3L_8$, $^3M_8$, $^1L_8$, $^1L_8$, $^3L_9$, $^3M_9$, $^3M_{10}$, $^1N_{10}$

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

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

$\underline{f}^{12}$ (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)

$^2F_{5/2}$, $^2F_{7/2}$

In **qlanth** this basis of levels can be retrieved through the function `BasisLSJ`. This is the ordered basis used for all intermediate coupling calculations.

```
BasisLSJ::usage="BasisLSJ[numE] returns the intermediate coupling
    basis L-S-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}.
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];
  )
];
```

1.2 More quantun numbers

Besides using an integer that simply 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 wavefunctions find in the groups $\mathcal{SO}(7)$ and $\mathcal{G}_2$.

1.2.1 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 consequency 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 the arrive going up the configurations.

There is, however, a much 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. This is a consequence of the Pauli 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 an useful theoretical construct, but the method of fractional parentage (which is what is implemented in **qlanth** is well suited also, 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.

1.2.2 $\mathcal{U}$ and $\mathcal{W}$

Much as L tells us how a rotation acts on a L wavefunction by mixing different M_L components, these other two labels specify how the wavefunctions transform under the operations of these 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 $\mathcal{C}(\mathcal{SO}(7))$ and $\mathcal{C}(\mathcal{G}_2)$ operators.

1.3 The $|LSJ\rangle$ intermediate coupling basis

When only interactions with spherical symmetry are kept (all but the crystal field or external magnetic field) then the total angular momentum J is a good quantum number and all the M_J projections are degenerate. This allows for a much more frugal description of the eigenstates, in what is traditionally called the intermediate coupling description. Not only the number of basis states that need to be considered is much less than otherwise, but also the diagonalization is more efficient since it can be carried out within subspaces with shared J.

In **qlanth** the function BasisLSJ can be used to retrieve the ordered basis that is used in the intermediate coupling description. This function admits the option "ReturnAsAssociation" in which the function returns an association where the keys are values of J and the values are lists of lists of the form {LS string, J}.

To obtain the blocks (indexed by J) representing the intermediate coupling Hamiltonian, the function SimplerSymbolicIntermediateHamMatrix is included in **qlanth**.

```
SimplerSymbolicIntermediateHamMatrix::usage = "
    SimplerSymbolicIntermediateHamMatrix[numE] is provides a variation
     of HamMatrixAssembly that returns the intermediate Hamiltonian
    blocks applying a simplifier. 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
    in the intermediate Hamiltoniann description. 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[SimplerSymbolicIntermediateHamMatrix] = {
  "Export" -> True,
  "PrependToFilename" -> "",
  "Overwrite" -> False,
  "Return" -> True,
  "Simplifier" -> Join[
    {F0,\[Sigma]SS},
    cfSymbols,
    TSymbols,
    casimirSymbols,
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
  ]
};
```

```
SimplerSymbolicIntermediateHamMatrix[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"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
    numE]<>"-"<>ToString[hash]<>".m"}];
    fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
    numE]<>"-"<>ToString[hash]<>".mx"}];
    If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue["
    Overwrite"]],
    (
      If[OptionValue["Return"],
      (
        Which[FileExistsQ[fnamemx],
        (
          Print["File ",fnamemx," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
          thisHamAssoc=Import[fnamemx];
          Return[thisHamAssoc];
        ),
        FileExistsQ[fname],
        (
          Print["File ",fname," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
          thisHamAssoc=Import[fname];
          Print["Exporting to file ",fnamemx," for quicker loading."
    ];
          Export[fnamemx,thisHamAssoc];
          Return[thisHamAssoc];
        )
        ]
      ),
      (
        Print["File ",fname," already exists, skipping ..."];
        Return[Null];
      )
      ]
    )
    ];
    Js           = AllowedJ[numE];
    thisHamAssoc = HamMatrixAssembly[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"],
      (
        Print["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 intermediate coupling description. To this aim the following functions in **qlanth** are instrumental: EigenLever, FreeHam, ListRepeater, and ListLever.

The function IntermediateSolver can be used to facilitate the calculation of a specific intermediate coupling level structure given values for the parameters that are kept in the description.

```
IntermediateSolver::usage="IntermediateSolver[numE, params] puts
    together (or retrieves from disk) the symbolic intermediate
    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 subpsace, 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 two elements. The first element is the global
    intermediate coupling basis for the f^numE configuration, given as
     a list of lists of the form {LS string, J}. The second 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. 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 intermediate coupling
    description.
```

```
";
Options[IntermediateSolver] = {
  "Overwrite Hamiltonian" -> False,
  "Return as states" -> True,
  "Simplifier" -> Join[
    cfSymbols,
    TSymbols,
    casimirSymbols,
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
  ],
  "PrintFun" -> PrintTemporary
};
IntermediateSolver[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["> IntermediateSolver for ",ln," with ",numE," f-
    electrons."];
    PrintFun["> Loading the symbolic intermediate coupling
    Hamiltonian ..."];
    simpleHam   = SimplerSymbolicIntermediateHamMatrix[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,"Print"->True];
    PrintFun[params];
    (* Create the numeric hamiltonian *)
    PrintFun["> Replacing parameters in the J-blocks of the
    intermediate coupling Hamiltonian to produce numeric arrays ..."];
    numHam     = N /@ Map[ReplaceInSparseArray[#, params]&, simpleHam
    ];
    Clear[simpleHam];
    (* Eigensolver *)
    PrintFun["> Diagonalizing the numerical Hamiltonian within each
    separat 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
    global intermediate 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];
        Return[{globalBasis, states}];
      ),
      Return[{basis, eigensys}]
    ];
  )
];
```

2 The coefficients of fractional parentage

In the 1920s and 1930s, when spectroscopic evidence was being studied to elucidate the principles of 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 $\underline{\mathrm{f}}^{n-1}$ 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

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

More importantly for **qlanth**, the coefficients of fractional parentage can be used to evaluate matrix elements of operators, such as in &qn-28, &qn-50, &qn-63, and &qn-40. 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 $\underline{\mathrm{f}}^{n}$ configurations, reduced matrix elements in $\underline{\mathrm{f}}^{3}$ for the three-body operators, and reduced matrix elements in $\underline{\mathrm{f}}^{2}$ 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 $\underline{\mathrm{f}}^{n}$ configurations.

In **qlanth** the function `GenerateCFPTable` is used to parse the data contained in this file. From this data an association `CFP` is generated, whose keys are made to represent LS terms from a

configuration $\underline{f}^n$ and whose values are lists which contain all the parents 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]

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

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

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]:

$$\langle \underline{\mathrm{f}}^{14-n}\alpha SL\|\hat{U}^{(K)}\|\underline{\mathrm{f}}^{14-n}\alpha' S'L'\rangle = -(-1)^K\langle \underline{\mathrm{f}}^{n}\alpha SL\|\hat{U}^{(K)}\|\underline{\mathrm{f}}^{n}\alpha' S'L'\rangle \quad (3)$$

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

$$\langle \underline{\mathrm{f}}^{14-n}\alpha SL\|\hat{V}^{(1K)}\|\underline{\mathrm{f}}^{14-n}\alpha' S'L'\rangle = (-1)^K\langle \underline{\mathrm{f}}^{n}\alpha SL\|\hat{V}^{(1K)}\|\underline{\mathrm{f}}^{n}\alpha' S'L'\rangle \quad (4)$$

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

3 The JJ' 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 $\underline{\mathrm{f}}^n$ 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 rank-2 arrays.

$\mathcal{F}$ig-1 shows roughly this block structure for $\underline{\mathrm{f}}^2$. 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).

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

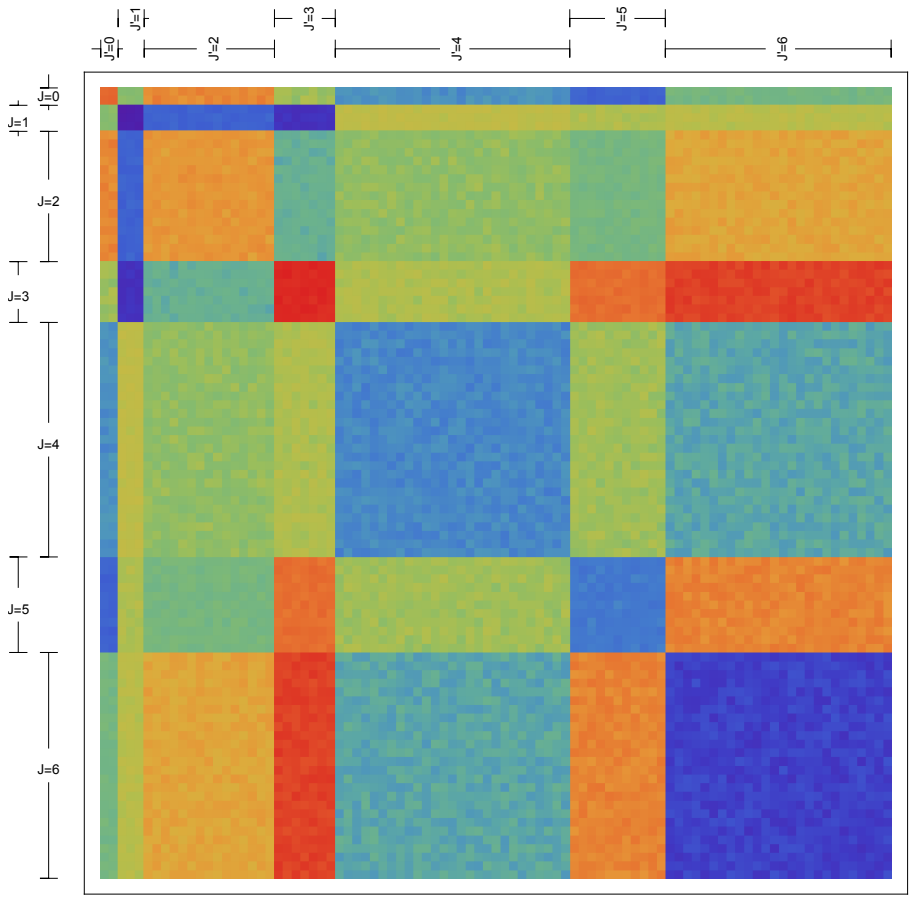

Figure 1: The block structure for $\underline{f}^2$

```
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[{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 `HamMatrixAssembly` takes them, assembles the arrays in block form, and finally flattens it to provide a rank-2 array. This 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 intermediate coupling description.

```
HamMatrixAssembly::usage="HamMatrixAssembly[numE] returns the
    Hamiltonian matrix for the f^n_i configuration. The matrix is
    returned as a SparseArray.
The function admits an optional parameter \"FilenameAppendix\" which
    can be used to modify the filename to which the resulting array is
     exported to.
It also admits an optional parameter \"IncludeZeeman\" which can be
    used to include the Zeeman interaction.
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[HamMatrixAssembly] = {
      "FilenameAppendix"->"",
      "IncludeZeeman"->False,
      "Set t2Switch"->True,
      "ReturnInBlocks"->False};
HamMatrixAssembly[nf_, OptionsPattern[]]:=Module[
  {numE, ii, jj, howManyJs, Js, blockHam},
  (
    (*######################################*)
    ImportFun = ImportMZip;
    (*######################################*)
    (*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, T11, 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.*)
    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 The effective Hamiltonian

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

To describe the effect of the charges in the lattice surrounding the lattice, the crystal field is introduced. In the simplest of embodiments, the crystal field is simply seen as the electrostatic field due to the surrounding charges. This model, however, has some limitations, but it gives way to a much broader validity based solely on symmetry arguments.

This framework sufficiently describes the interactions within a free ion. However, to extend this model to ions within a crystal, one incorporates this through what is called the crystal field. This is often achieved by considering the electric field that an ion experiences from the surrounding charges in the crystal lattice, a concept referred to as the crystal field effect.

The Hilbert space of a multi-electron ion is a vast stage. In principle the Hilbert space should have a countable infinity of discrete states and an uncountable infinity of states to describe the 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 neodymium atom. In principle, this gives us the daunting task of dealing with 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 fifty seven. 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 here will all be considered to be located in f-orbitals, or what is the same, that they are described by $\underline{f}^n$wavefunctions. This is, however, a harsh approximation, but thank-

fully one can make some amends to it. The effects that arise in the single configuration description because of omitting all the other possible orbitals where the electrons might find themselves, this is what is called *configuration-interaction*.

These effects can be brought within the simplified description only through the help of 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' [Rud07] book on theoretical atomic spectroscopy or this article [Lin74] by Lindgren. What results from this are operators that now act solely within the single configuration but with a convoluted 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.

The coefficients that result in the Hamiltonian one could try to evaluate, however within the **semi-empirical** approach these parameters are left to be fitted against experimental data, and perhaps 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 model Hamilonian that is fitted is based on a clear physical picture inherited from atomic physics.

Putting all of this together leads to the following Hamiltonian. In there, "v-electrons" is shorthand for valence electrons.

$$\hat{\mathcal{H}} = \underbrace{\hat{\mathcal{H}}_{\mathrm{k}}}_{\text{kinetic}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:sn}}}_{\text{e:shielded nuc}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:e}}}_{\text{e:e}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:o}}}_{\text{spin-orbit}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:s}}}_{\substack{\text{spin:spin}\\ \text{and spin:other-orbit}}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:oo\oplus ecs:o}}}_{\substack{\text{spin:other-orbit}\\ \text{ec-correlated-spin:orbit}}} + \tag{5}$$

$$\underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{G_2}}}_{\mathrm{G_2}\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(7)}}_{\mathcal{SO}(7)\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\text{Y}}}_{\substack{\text{effective}\\ \text{three-body}}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{cf}}}_{\text{crystal field}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{Z}}}_{\text{Zeeman}} \tag{6}$$

$$\hat{\mathcal{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m_e}\sum_{i=1}^{n}\nabla_i^2 \text{ (kinetic energy of } n \text{ v-electrons)} \tag{7}$$

$$\hat{\mathcal{H}}_{\mathrm{e:sn}} = \sum_{i=1}^{n} V_{\mathrm{sn}}(r_i) \text{ (interaction of v-electrons with shielded nuclear charge)} \tag{8}$$

$$\hat{\mathcal{H}}_{\mathrm{e:e}} = \sum_{i>j}^{n,n}\frac{e^2}{\|\vec{r}_i - \vec{r}_j\|} = \sum_{k=0,2,4,6} F^{(k)}\hat{f}_k \text{ (v-electron:v-electron repulsion)} \tag{9}$$

$$\hat{\mathcal{H}}_{\mathrm{s:o}} = \begin{cases} \sum_{i=1}^{n}\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \text{ with } \xi(r_i) = \frac{\hbar^2}{2m^2c^2r_i}\frac{\mathrm{d}V_{\mathrm{sn}}(r_i)}{\mathrm{d}r_i} \\ \sum_{i=1}^{n}\zeta\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \text{ with } \zeta \text{ the radial average of } \xi(r_i) \text{ or used as phenomenological parameter} \end{cases} \tag{10}$$

$$\hat{\mathcal{H}}_{\mathrm{s:s}} = \sum_{k=0,2,4} M^{(k)}\hat{m}_k^{ss} \tag{11}$$

$$\hat{\mathcal{H}}_{\mathrm{s:oo\oplus ecs:o}} = \sum_{k=2,4,6} P^{(k)}\hat{p}_k + \sum_{k=0,2,4} M^{(k)}\hat{m}_k \tag{12}$$

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

$$\hat{\mathcal{H}}_{\mathcal{SO}(3)} = \alpha\,\mathcal{C}(\mathcal{SO}(3)) = \alpha\hat{L}^2 \text{ (Trees effective operator)} \tag{13}$$

$$\hat{\mathcal{H}}_{\mathrm{G_2}} = \beta\,\mathcal{C}(\mathcal{G}_2) \tag{14}$$

$$\hat{\mathcal{H}}_{\mathcal{SO}(7)} = \gamma\,\mathcal{C}(\mathcal{SO}(7)) \tag{15}$$

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

$$\hat{\mathcal{H}}_{\mathrm{cf}} = \sum_{i=1}^{n} V_{\mathrm{CF}}(\hat{r}_i) = \sum_{i=1}^{n}\sum_{k=2,4,6}\sum_{q=-k}^{k}\mathcal{B}_q^{(k)}\mathcal{C}_q^{(k)}(i) \text{ (crystal field interaction of v-electrons with electrostatic field due to surroundings)} \tag{17}$$

$$\hat{\mathcal{H}}_{\mathrm{Z}} = -\vec{B}\cdot\hat{\mu} = \mu_B\vec{B}\cdot\left(\hat{L} + g_s\hat{S}\right) \text{ (interaction with a magnetic field)} \tag{18}$$

It is of some importance 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 has been already integrated in the parameters that the Hamiltonian has.

4.1 $\hat{\mathcal{H}}_{\mathrm{k}}$: kinetic energy

$$\hat{\mathcal{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 \text{ (kinetic energy of N v-electrons)} \tag{19}$$

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 Hamiltonian, tt might be instructive, however, to note that this term imparts an energy of about $10\,\mathrm{eV} = 10^6\mathrm{K}$ to each electron

4.2 $\hat{\mathcal{H}}_{\mathbf{e:sn}}$: the central field potential

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 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 literally like spherical shells surrounding the nucleus.

$$\hat{\mathcal{H}}_{\mathrm{e:sn}} = -e^2 \sum_{i=1}^{Z} \frac{1}{r_i} + e^2 \underbrace{\sum_{i=1}^{n} \sum_{j=1}^{Z-n} \frac{1}{r_{ij}}}_{\substack{\text{Repulsion between valence} \\ \text{and inner shell} \\ \text{electrons}}} \approx \sum_{i=1}^{n} V_{\mathrm{sn}}(r_i) \text{ (with Z = atomic No.)} \tag{20}$$

The precise form of $V_{\mathrm{sn}}(r_i)$ 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.

4.3 $\hat{\mathcal{H}}_{\mathbf{e:e}}$: e:e repulsion

$$\hat{\mathcal{H}}_{\mathrm{e:e}} = \sum_{i>j}^{n,n} \frac{e^2}{\|\vec{r}_i - \vec{r}_j\|} = \sum_{k=0,2,4,6} F^{(k)} \hat{f}_k = \sum_{k=0,1,2,3} E_k \hat{e}^k \tag{21}$$

This term is the first we will not discard. Calculating this term for the $\underline{\mathrm{f}}^n$configurations was one of the contribution 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. The insight that Racah had was that if in a given operator one identified the parts in it that transformed nicely according to the different symmetry groups present in the problem, then calculating the necessary matrix element in all $\underline{\mathrm{f}}^n$configurations can be greatly simplified.

The functions used in `qlanth` to compute these LS-reduced matrix elements 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 $\underline{\mathrm{f}}^n$configuration. By default this term is computed in terms of $F^{(k)}$ Slater integrals, but it can also be computed in of the E_k Racah parameters, the functions `EtoF` and `FtoE` instrumental for going from one representation to the other.

$$\langle \underline{\mathrm{f}}^n \alpha^{2S+1}\mathrm{L} \| \hat{\mathcal{H}}_{\mathrm{e:e}} \| \underline{\mathrm{f}}^n \alpha'^{2S'+1}\mathrm{L'} \rangle = \sum_{k=0,2,4,6} F^{(k)} f_k(n, \alpha LS, \alpha' L' S') \tag{22}$$

where

$$f_k(n,\alpha LS,\alpha' L' S') = \frac{1}{2}\delta(S,S')\delta(L,L')\langle \underline{\mathrm{f}}\|\hat{C}^{(k)}\|\underline{\mathrm{f}}\rangle^2 \times$$
$$\left\{ \frac{1}{2L+1}\sum_{\alpha'' L''} \langle \underline{\mathrm{f}}^n \alpha'' L'' S\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha LS\rangle\langle \underline{\mathrm{f}}^n \alpha'' L'' S\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha' LS\rangle - \delta(\alpha,\alpha')\frac{n\,(4\underline{\mathrm{f}}+2-n)}{(2\underline{\mathrm{f}}+1)(4\underline{\mathrm{f}}+1)} \right\} \quad (23)$$

```
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];
  )
];
```

```
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 / TPO[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}];
  )
];
```

4.4 $\hat{\mathcal{H}}_{\text{s:o}}$: 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

$$\hat{h}_{\text{s:o}} = \frac{\hbar^2}{2m_e^2c^2}\left(\frac{1}{r}\frac{\mathrm{d}V_{\text{s:n}}}{\mathrm{d}r}\right)\hat{l}\cdot\hat{s} \coloneqq \zeta(r)\hat{l}\cdot\hat{s}. \tag{24}$$

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

$$\hat{\mathcal{H}}_{\text{s:o}} = \sum_i^n \zeta\hat{l}_i\cdot\hat{s}_i. \tag{25}$$

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

$$\begin{aligned}\langle\alpha LSJM_J|\hat{\mathcal{H}}_{\text{s:o}}|\alpha' L'S'J'M_{J'}\rangle &= \zeta\delta(J,J')\delta(M_J,M_{J'})\langle\alpha LSJM_J|\sum_i^n \hat{l}_i\cdot\hat{s}_i|\alpha' L'S'JM_J\rangle \\ &= \zeta\delta(J,J')\delta(M_J,M_{J'})(-1)^{J+L+S'}\begin{Bmatrix} L & L' & 1 \\ S' & S & J\end{Bmatrix}\langle\alpha LS|\sum_i^n \hat{l}_i\cdot\hat{s}_i|\alpha' L'S'\rangle \\ = \zeta\delta(J,J')\delta(M_J,M_{J'})(-1)^{J+L+S'}&\begin{Bmatrix} L & L' & 1 \\ S' & S & J\end{Bmatrix}\sqrt{\underline{\ell}(\underline{\ell}+1)(2\underline{\ell}+1)}\langle\alpha LS\|\hat{V}^{(11)}\|\alpha' L'S'\rangle.\end{aligned} \tag{26}$$

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

$$\hat{V}^{(11)} = \sum_{i=1}^n \left(\hat{s}\hat{u}^{(1)}\right)_i, \tag{27}$$

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

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

$$\begin{aligned}\langle\underline{\ell}^n\psi\|\hat{V}^{(1k)}\|\underline{\ell}^n\psi'\rangle = \langle\underline{\ell}^n\alpha LS\|\hat{V}^{(1k)}\|\underline{\ell}^n\alpha' L'S'\rangle = n\sqrt{\underline{\jmath}(\underline{\jmath}+1)(2\underline{\jmath}+1)}\sqrt{[S][L][S'][L']}\times \\ \sum_{\bar{\psi}}(-1)^{\bar{S}+\bar{L}+S+L+\underline{\ell}+\underline{\jmath}+k+1}\left(\psi\left\{|\bar{\psi}\right)\left(\bar{\psi}\right\}\psi'\right)\begin{Bmatrix} S & S' & 1 \\ \underline{\jmath} & \underline{\jmath} & \bar{S}\end{Bmatrix}\begin{Bmatrix} L & L' & k \\ \underline{\ell} & \underline{\ell} & \bar{L}\end{Bmatrix}\end{aligned} \tag{28}$$

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 f-electron configurations, we have $\underline{\ell} = 3$ and $\underline{s} = \frac{1}{2}$ as is due to the electron.

```
ReducedV1k::usage = "ReducedV1k[n, l, 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];
  )
```

```
];
```

4.5 $\hat{\mathcal{H}}_{\mathcal{SO}(3)}, \hat{\mathcal{H}}_{\mathbf{G}_2}, \hat{\mathcal{H}}_{\mathcal{SO}(7)}$: electrostatic configuration interaction

This is a first term 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 $\underline{\mathrm{f}}^n$ 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 (MII) and the third spectrum of iron (FeIII).

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 $\hat{\mathcal{H}}_{\mathrm{G}_2}$ 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 $\hat{\mathcal{H}}_{\mathcal{SO}(3)}$, $\hat{\mathcal{H}}_{\mathrm{G}_2}$ and $\hat{\mathcal{H}}_{\mathcal{SO}(7)}$ are diagonal in all quantum numbers and are given by

$$\langle \underline{\ell}^n \alpha S L J M_J | \hat{\mathcal{H}}_{\mathcal{SO}(3)} | \underline{\ell}^n \alpha' S' L' J' M_J' \rangle = \alpha \delta(\alpha S L J M_J, \alpha' S' L' J' M_J') L(L+1) \tag{29}$$

$$\langle \underline{\ell}^n U \alpha S L J M_J | \hat{\mathcal{H}}_{\mathrm{G}_2} | \underline{\ell}^n U \alpha' S' L' J' M_J' \rangle = \beta \delta(\alpha S L J M_J, \alpha' S' L' J' M_J') \Lambda_{\mathcal{G}_2}(U) \tag{30}$$

$$\langle \underline{\ell}^n W \alpha S L J M_J | \hat{\mathcal{H}}_{\mathcal{SO}(7)} | \underline{\ell}^n W \alpha' S' L' J' M_J \rangle = \gamma \delta(\alpha S L J M_J, \alpha' S' L' J' M_J') \Lambda_{\mathcal{SO}(7)}(W) \tag{31}$$

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[{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[{SL_, SpLp_}]:=Module[
  {S, L, val},
  (
    {S, L} = FindSL[SL];
    val = (
      If[SL == SpLp,
        CasimirSO3[{SL, SL}] +
        CasimirSO7[{SL, SL}] +
        CasimirG2[{SL, SL}],
        0
      ]
      );
    ElectrostaticConfigInteraction[{S, L}] = val;
    Return[val];
  )
];
```

4.6 $\hat{\mathcal{H}}_{\text{s:s-s:oo}}$: *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 $\underline{\text{f}}^2$ or in $\underline{\text{f}}^3$ and then we "pull" them up for all $\underline{\text{f}}^n$ configuration with the help of formulas involving 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 as a relativistic correction with the Dirac equation as the starting point. The multi-electron spin-orbit effects can be derived from the Breit operator [BS57] which is added to the relativistic description of a many-particle system in order to account for retardation of the electromagnetic field

$$\hat{\mathcal{H}}_B = -\frac{1}{2}e^2 \sum_{i>j} \left[(\alpha_i \cdot \alpha_j) \frac{1}{r_{ij}} + (\alpha_i \cdot \vec{r}_{ij})(\alpha_j \cdot \vec{r}_{ij}) \frac{1}{r_{ij}^3} \right]. \tag{32}$$

When this operator is expanded in powers of v/c, a number of non-relativistic inter-electron interactions result. Two of them being 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

$$M^{(k)} := \frac{e^2\hbar^2}{8m^2c^2} \langle (nl)^2 | \frac{r_<^k}{r_>^{k+3}} | (nl)^2 \rangle \tag{33}$$

With these, the expression for the *spin-spin* term is [JCC68]

$$\hat{\mathcal{H}}_{s:s} = -2 \sum_{i\neq j} \sum_k M^{(k)} \sqrt{(k+1)(k+2)(2k+3)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle \langle \underline{\ell} \| C^{(k+2)} \| \underline{\ell} \rangle \left\{ \hat{w}_i^{(1,k)} \hat{w}_j^{(1,k+2)} \right\}^{(2,2)0} \tag{34}$$

and the one for *spin-other-orbit*

$$\begin{aligned} \hat{\mathcal{H}}_{s:oo} = \sum_{i\neq j} \sum_k & \sqrt{(k+1)(2\underline{\ell}+k+2)(2\underline{\ell}-k)} \times \\ & \left[\left\{ \hat{w}_i^{(0,k+1)} \hat{w}_j^{(1,k)} \right\}^{(11)0} \left\{ M^{(k-1)} \langle \underline{\ell} \| C^{(k+1)} \| \underline{\ell} \rangle^2 + 2M^{(k)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle^2 \right\} + \right. \\ & \left. \left\{ \hat{w}_i^{(0,k)} \hat{w}_j^{(1,k+1)} \right\}^{(11)0} \left\{ M^{(k)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle^2 + 2M^{(k-1)} \langle \underline{\ell} \| C^{(k+1)} \| \underline{\ell} \rangle^2 \right\} \right]. \end{aligned} \tag{35}$$

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

$$\langle \underline{\ell} \| \hat{w}^{(\kappa,k)} \| \underline{\ell} \rangle = \sqrt{[\![\kappa]\!][\![k]\!]} \tag{36}$$

The complexity of the above expressions for can be identified 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

$$\sqrt{5}\hat{\mathcal{T}}_0^{(22)} := \hat{\mathcal{H}}_{\text{s:s}} \tag{37}$$

$$-\sqrt{3}\hat{\mathcal{T}}_0^{(11)} := \hat{\mathcal{H}}_{\text{s:oo}}. \tag{38}$$

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

$$\langle \gamma SLJ | \hat{\mathcal{H}} | \gamma' S'L'J' \rangle = \delta(J,J') \begin{Bmatrix} S' & L' & J \\ L & S & t \end{Bmatrix} \langle \gamma SL \| \hat{\mathcal{T}}^{(tt)} \| \gamma' S'L' \rangle. \tag{39}$$

This above relationship is 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 $\underline{\mathrm{f}}^{n-1}$ through equation 4 in Judd et al [JCC68]

$$\langle \underline{\mathrm{f}}^n \psi \| \hat{\mathcal{T}}^{(tt)} \| \underline{\mathrm{f}}^n \psi' \rangle = \frac{n}{n-2} \sum_{\bar{\psi},\bar{\psi}'} (-1)^{\bar{S}+\bar{L}+\underline{\jmath}+\underline{\ell}+S'+L'} \sqrt{[\![S]\!][\![S']\!][\![L]\!][\![L']\!]} \times \left(\psi\{\|\bar{\psi}\right)\left(\psi'\{\|\bar{\psi}'\right) \begin{Bmatrix} S & t & S' \\ \bar{S}' & \underline{\jmath} & \bar{S} \end{Bmatrix} \begin{Bmatrix} L & t & L' \\ \bar{L}' & \underline{\ell} & \bar{L} \end{Bmatrix} \langle \underline{\mathrm{f}}^{n-1} \bar{\psi} \| \hat{\mathcal{T}}^{(tt)} \| \underline{\mathrm{f}}^{n-1}] \bar{\psi}' \rangle. \quad (40)$$

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 &qn-39 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 $\hat{\mathcal{H}}_{\mathrm{s:oo}}$, since this term is traditionally computed (with a slight modification) at the same 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 the
    operator corresponding to the inter-electron between spin.
It does this 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[TP0[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]
    ]
  )
];
```

4.7 $\hat{\mathscr{H}}_{\text{ecs:o}}$: *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])

$$\hat{\mathscr{H}}_{\text{ci}} = -\sum_{\chi}\sum_{i}\frac{1}{E_\chi}\xi(r_i)\left(\hat{\underline{\jmath}}_i\cdot\hat{\underline{\ell}}_i\right)|\chi\rangle\langle\chi|\hat{\mathfrak{C}} - \frac{1}{E_\chi}\hat{\mathfrak{C}}|\chi\rangle\langle\chi|\xi(r_i)\left(\hat{\underline{\jmath}}_i\cdot\hat{\underline{\ell}}_i\right) \tag{41}$$

where $\xi(r_h)(\hat{\underline{\jmath}}_h\cdot\hat{\underline{\ell}}_h)$ 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 to which one is "interacting" with. 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.

$$\hat{\mathcal{H}}_{\text{ci}} = -\sqrt{3}\,\hat{t}_0^{(11)} \tag{42}$$

Judd *et al.* 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

$$P^{(k)} = 6\sum_{f'} \frac{\zeta_{ff'}}{E_{ff'}} R^{(k)}(ff, ff') \text{ for } k = 0, 2, 4, 6. \tag{43}$$

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

$$\zeta_{ff'} \coloneqq \langle f|\xi(r)|f'\rangle \tag{44}$$

$$R^{(k)}(ff, ff') \coloneqq e^2\langle f_1 f_2|\frac{r_<^k}{r_>^{k+1}}|f_1 f_2'\rangle. \tag{45}$$

In the semi-empirical approach embodied by **qlanth**, calculating these quantities *ab-initio* is not the objective, rendering the precise definition of these parameters non-essential. Nonetheless, these expressions frequently serve to justify the ratios between different orders of these quantities. 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

$$\langle\psi\|\hat{\mathcal{T}}^{(11)} + \hat{t}^{(11)}\|\psi'\rangle = \sum a_i\langle\psi\|\hat{z}_i\|\psi'\rangle. \tag{46}$$

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

$$\hat{\mathcal{H}}_{\text{s:oo}} + \hat{\mathcal{H}}_{\text{ecs:o}} = \hat{\mathcal{T}}^{(11)} + \hat{t}^{(11)} - \frac{1}{6}a_{13}\hat{z}_{13} \tag{47}$$

where

$$a_{13} = -33M^{(0)} + 3M^{(2)} + \frac{15}{11}M^{(4)} - 6P^{(0)} + \frac{3}{2}\left(\frac{35}{225}P^{(2)} + \frac{77}{1089}P^{(4)} + \frac{25}{1287}P^{(6)}\right). \tag{48}$$

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 pre-computed 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, SLJ, SLJp, J}] returns the matrix element of the magnetic interaction between the terms SLJ and SLJp in the f^numE configuration. 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, SLJ, SLJp, J}].
The part corresponding to SOO and ECSO is provided by the function SOOandECSO[{numE, SLJ, SLJp, J}].
The function requires chenDeltas to be loaded into the session.
The option \"ChenDeltas\" can be used to include or exclude the Chen deltas from the calculation. The default is to exclude them.";
Options[MagneticInteractions] = {"ChenDeltas" -> False};
MagneticInteractions[{numE_, SLJ_, SLJp_, J_}, OptionsPattern[]] :=
  (
    key = {numE, SLJ, SLJp, 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, SLJ, SLJp}],
      (
        {S, L} = FindSL[SLJ];
        {Sp, Lp} = FindSL[SLJp];
        phase   = Phaser[Sp + L + J];
        Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
        Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
        {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SLJ, SLJp}]["wrong"];
        total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
        total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
        total  = total /. Prescaling;
        total  = wChErrA * total + (1 - wChErrA) * (ss + sooandecso)
      )
    ];
    (* In the type B errors the wrong values are zeros all around *)
    If[MemberQ[chenDeltas["B"], {numE, SLJ, SLJp}],
      (
        {S, L} = FindSL[SLJ];
        {Sp, Lp} = FindSL[SLJp];
        phase = Phaser[Sp + L + J];
        Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
        Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
        {M0v, M2v, M4v, P2v, P4v, P6v} = {0, 0, 0, 0, 0, 0};
        total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
        total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
        total  = total /. Prescaling;
        total  = wChErrB * total + (1 - wChErrB) * (ss + sooandecso)
```

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

```
GenerateSpinOrbitTable::usage = "GenerateSpinOrbitTable[nmax]
    computes the matrix values 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 subfolder for
     the folder in which this package is in. 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"],
      (
        Print["Exporting to file "<>ToString[exportFname]];
        Export[exportFname, SpinOrbitTable];
      )
    ];
    Return[SpinOrbitTable];
  )
];
```

```
GenerateSOOandECSOTable::usage="GenerateSOOandECSOTable[nmax]
    generates the 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 the double tensor $\hat{\mathcal{T}}^{(22)}$ which are stored in the association T22Table.

```
GenerateSpinSpinTable::usage="GenerateSpinSpinTable[nmax] generates
    the 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 {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-40 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 the
    operator corresponding to the inter-electron between spin.
It does this 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-47 as stored in the association SOOandECSOLSTable.

```
GenerateSOOandECSOTable::usage="GenerateSOOandECSOTable[nmax]
    generates the 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 ℰqn-40 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 1966 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.
```

```
Rigurously 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];
  )
];
```

4.8 $\hat{\mathcal{H}}_{\lambda}$: three-body effective operators

The three-body operators arise in the Hamiltonian 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 ($\hat{t}$ fron *t*hree)

$$\hat{\mathcal{H}}_{\lambda} = T'^{(2)}\hat{t}'_2 + T'^{(11)}\hat{t}'_{11} \sum_{\substack{k=2,3,4,6,7,8,\\ 11,12,14,15,\\ 16,17,18,19}} T^{(k)}\hat{t}_k, \tag{49}$$

where $\hat{t}'_2$ and $\hat{t}'_{11}$ are operators that were initially included but which were later replaced 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).

Furtermore, 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 as in equation 4 of [Jud66] (adding the explicit dependence on J and M_J):

$$\langle \underline{f}^n \psi | \hat{t}_i | \underline{f}^n \psi' \rangle = \delta(J, J')\delta(M_J, M'_J) \frac{n}{n-3} \sum_{\bar{\psi}\bar{\psi}'} \left(\psi \left\{ \right| \bar{\psi} \right) \left(\psi' \left\{ \right| \bar{\psi}' \right) \langle \underline{f}^{n-1} \bar{\psi} | \hat{t}_i | \underline{f}^{n-1} \bar{\psi}' \rangle. \tag{50}$$

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 $\hat{t}_{11}$ for which primed variants $\hat{t}'_2$ and $\hat{t}'_{11}$ are calculated differently beyond the half filled shell. In the case of the other 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 $\hat{t}'_2$ is used then $\hat{t}_2$ 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 parsimony, 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 $\underline{f}^7$.

```
GenerateThreeBodyTables::usage="This function generates the matrix
    elements for the three body operators using the coefficients of
    fractional parentage, including those beyond f^7.";
Options[GenerateThreeBodyTables] = {"Export" -> False};
GenerateThreeBodyTables[nmax_Integer : 14, OptionsPattern[]] := (
  tiKeys = {"t_{2}", "t_{2}^{'}", "t_{3}", "t_{4}", "t_{6}", "t_{7}",
    "t_{8}", "t_{11}", "t_{11}^{'}", "t_{12}", "t_{14}", "t_{15}",
    "t_{16}", "t_{17}", "t_{18}", "t_{19}"};
  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 matrix elements of ti in f^n by using the
  matrix elements in f3 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 matrix elements of t^i for n up to nmax*)
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, nmax}
  ];

  (* Now use those 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}];
);

ScalarOperatorProduct::usage="ScalarOperatorProduct[op1, op2, numE]
    calculated the innerproduct between the two scalar operators op1
    and op2.";
ScalarOperatorProduct[op1_, op2_, numE_]:=Module[
  {terms, S, L, factor, term1, term2},
  (
    terms = AllowedNKSLTerms[numE];
    Simplify[
      Sum[(
        {S, L} = FindSL[term1];
        factor = TPO[S, L];
        factor * op1[{term1, term2}] * op2[{term2, term1}]
        ),
      {term1, terms},
      {term2, terms}
      ]
    ]
  )
];
```

```
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 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 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];
);
```

4.9 $\hat{\mathcal{H}}_{\mathbf{cf}}$: 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:

$$V(r_i, \theta_i, \phi_i) = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{A}_q^{(k)} r_i^k C_q^{(k)}(\theta_i, \phi_i) \tag{51}$$

Where 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 since here 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

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=-k}^{k} \mathcal{A}_q^{(k)} r_i^k C_q^{(k)}(\theta_i, \phi_i). \tag{52}$$

And if we average the radial coordinate,

$$\hat{\mathcal{H}}_{\mathrm{cf}} = \sum_{i=1}^{n} \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{B}_q^{(k)} C_q^{(k)}(i) \tag{53}$$

where the radial average is included as

$$\mathcal{B}_q^{(k)} := \mathcal{A}_q^{(k)} \langle 4f | r^k | 4f \rangle. \tag{54}$$

$\mathcal{B}_q^{(k)}$ may be complex in general. However, since the sum in Eqn-52 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 $C_q^{(k)}$ under complex conjugation is considered, $C_q^{(k)*} = (-1)^q C_{-q}^{(k)}$, it is necessary that

$$\mathcal{B}_q^{(k)} = (-1)^q \mathcal{B}_{-q}^{(k)*}. \tag{55}$$

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

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=0}^{k} B_q^{(k)} \left(C_q^{(k)} + (-1)^q C_{-q}^{(k)} \right) + iS_q^{(k)} \left(C_q^{(k)} - (-1)^q C_{-q}^{(k)} \right). \tag{56}$$

A staple of the Wigner-Racah algebra is writing up operators on 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):

$$\langle \ell \| \hat{u}^{(k)} \| \ell' \rangle = 1. \tag{57}$$

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

$$\hat{U}^{(k)} = \sum_{i}^{n} \hat{u}_i^{(k)}. \tag{58}$$

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

$$C_q^{(k)} = \langle \underline{\ell} \| C^{(k)} \| \underline{\ell}' \rangle \hat{u}_q^{(k)} = (-1)^{\underline{\ell}} \sqrt{[\![\underline{\ell}]\!][\![\underline{\ell}']\!]} \begin{pmatrix} \underline{\ell} & k & \underline{\ell}' \\ 0 & 0 & 0 \end{pmatrix} \hat{u}_q^{(k)}. \tag{59}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{\mathscr{H}}_{\rm cf} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-3}} = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathscr{B}_q^{(k)} \langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle \langle \underline{\ell} \| \hat{C}^{(k)} \| \underline{\ell} \rangle \tag{60}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' S'L'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-4}} = (-1)^{J-M_J} \begin{pmatrix} J & k & J' \\ -M_J & q & M_{J'} \end{pmatrix} \langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle \tag{61}$$

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

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Wybourne eqn. 6-5}} = (-1)^{S+L+J'+k} \sqrt{[\![J]\!][\![J']\!]} \times \\ \begin{Bmatrix} J & J' & k \\ L' & L & S \end{Bmatrix} \langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle. \end{aligned} \tag{62}$$

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

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Cowan eqn. 11.53}} = \delta(S,S') n (-1)^{\underline{\ell}+L+k} \sqrt{[\![L]\!][\![L']\!]} \times \\ \sum_{\bar{\alpha}\bar{L}\bar{S}} (-1)^{\bar{L}} \begin{Bmatrix} \underline{\ell} & k & \underline{\ell} \\ L & \bar{L} & L' \end{Bmatrix} \left(\underline{\ell}^n \alpha LS \left\{ \underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} \right) \left(\underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} \right\} \underline{\ell}^n \alpha' L'S' \right). \end{aligned} \tag{63}$$

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 $\langle \underline{\ell}^n \alpha SLJM_J | \hat{\mathscr{H}}_{\rm cf} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle$ have the same parity, then the overall parity of the braket is determined by the parity of $C_q^{(k)}$, and since the parity of $C_q^{(k)}$ 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:

$$\hat{\mathscr{H}}_{\rm cf}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=2,4,6} \sum_{q=0}^{k} B_q^{(k)} \left(C_q^{(k)} + (-1)^q C_{-q}^{(k)} \right) + i S_q^{(k)} \left(C_q^{(k)} - (-1)^q C_{-q}^{(k)} \right). \tag{64}$$

The above equations are implemented in **qlanth** by the function CrystalField. This function puts together the symbolic sum in &eqn-60 by using the function Cqk. Cqk then uses the diagonal reduced matrix elements of $C_q^{(k)}$ 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]
    gives the general expression for the matrix element of the crystal
     field Hamiltonian parametrized with Bqk and Sqk coefficients as a
     sum over spherical harmonics Cqk.
Sometimes this expression only includes Bqk coefficients, see for
    example eqn 6-2 in Wybourne (1965), but one may also split the
    coefficient into real and imaginary parts as is done here, in an
    expression that is patently Hermitian.";
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];
]
```

4.10 $\hat{\mu}$ 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

$$\hat{\mu} = -\mu_{\mathrm{B}} \left(\hat{L} + g_s \hat{S}\right)^{(1)}, \text{with } \mu_{\mathrm{B}} = 1/2. \tag{65}$$

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]

$$\langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle = \delta(\alpha LSJ, \alpha' L'S'J') \sqrt{J(J+1)(2J+1)} + \\ \delta(\alpha LS, \alpha' L'S') (-1)^{L+S+J+1} \sqrt{[\![J]\!][\![J]\!]} \begin{Bmatrix} L & S & J \\ 1 & J' & S \end{Bmatrix} \tag{66}$$

And then those reduced matrix elements are used to resolve the M_J components for $q = -1, 0, 1$ through Wigner-Eckart

$$\langle \alpha LSJM_J | \left(\hat{L} + g_s \hat{S}\right)^{(1)}_q | \alpha' L'S'J'M_{J'} \rangle = \\ (-1)^{J-M_J} \begin{pmatrix} J & 1 & J' \\ -M_J & q & M'_J \end{pmatrix} \langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle \tag{67}$$

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

$$\hat{\mu}_x = \frac{\hat{\mu}^{(1)}_{-1} - \hat{\mu}^{(1)}_{+1}}{\sqrt{2}} \tag{68}$$

$$\hat{\mu}_y = i\frac{\hat{\mu}^{(1)}_{-1} + \hat{\mu}^{(1)}_{+1}}{\sqrt{2}} \tag{69}$$

$$\hat{\mu}_z = \hat{\mu}^{(1)}_0 \tag{70}$$

```
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.";
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 `HamMatrixAssembly`. 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 pseudo-energy unit Kayser (cm^{-1}). The conversion factor is called `TeslaToKayser` in the file `qonstants.m`.

4.11 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-4 and &qn-3.

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

$$\zeta, T^{(2)}, T^{(3)}, T^{(4)}, T^{(6)}, T^{(7)}, T^{(8)}, B_q^{(k)} \tag{71}$$

In **qlanth** this symmetry is taken into account when the function `HamMatrixAssembly` is called, which uses `HoleElectronConjugation` to enforce the necessary sign changes.

```
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 complentary 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 the
    relationship is more complex and is controlled by the value of the
     isE variable.";
HoleElectronConjugation[params_] := Module[
  {newparams = params},
  (
    flipSignsOf = {ζ, T2, T3, T4, T6, T7, T8};
    flipSignsOf = Join[flipSignsOf, cfSymbols];
    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];
  )
];
```

5 Transitions

5.1 Magnetic Dipole Transitions

qlanth can also calculate 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])

$$\hat{S}(\psi, \psi') \coloneqq \left|\langle\psi|\hat{\mu}|\psi'\rangle\right|^2 = \left|\langle\psi|\hat{\mu}_x|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_y|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_z|\psi'\rangle\right|^2 \tag{72}$$

$^2F_{3/2}$

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 $\hat{\mathcal{S}}(\psi,\psi')$.

```
MagDipLineStrength::usage="MagDipLineStrength[theEigensys, numE]
    takes the eigensystem of an ion and the number numE of f-electrons
     that correspond to it and it 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) 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[
  {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,
      (
        Print["Invalid option for \"Units\". Options are \"SI\" and \
    "Hartree\"."];
        Abort[];
      )
    ];
  )
];
```

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

$$\mathrm{A_{MD}}\left(|\psi_i\rangle \rightsquigarrow |\psi_f\rangle\right) = \frac{16\pi^3\mu_0}{3h}\frac{n^3}{\lambda^3}\frac{\hat{S}(\psi_i,\psi_f)}{g_i}, \tag{73}$$

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 fine-grained description that **qlanth** uses, J is no longer a good quantum number so $g_i = 1$.

```
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 pseudo-energy unit kayser.*)
    transitionWaveLengthsInMeters = 0.01/energyDiffs;

    unitFactor     = Which[
```

```
    OptionValue["Units"]=="Hartree",
    (
      (* The bohrRadius factor in SI neede to convert the wavelengths
     which are assumed in m*)
      16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckFine)) * bohrRadius^3
    ),
    OptionValue["Units"]=="SI",
    (
      16 \[Pi]^3 \[Mu]0/(3 hPlanck)
    ),
    True,
    (
      Print["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 state 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 more frequent interest. The oscillator strength is given by [CFW65]

$$\mathrm{f}_{\mathrm{MD}}\left(|\psi_g\rangle \rightsquigarrow |\psi_e\rangle\right) = \frac{8\pi^2 m_e}{3\,hc\,e^2}\frac{n}{\lambda}\frac{\hat{S}(\psi_g,\psi_e)}{g_g} \tag{74}$$

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 GroundMDOscillatorStrength implements the calculation of the oscillator strengths from the ground state to all the excited ones.

```
GroundMDOscillatorStrength::usage="GroundMDOscillatorStrength[
    eigenSys, numE] calculates the magnetic diople 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 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[GroundMDOscillatorStrength]={"RefractiveIndex"->1};
GroundMDOscillatorStrength[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];
  )
];
```

5.2 Intermediate coupling

5.2.1 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 namely 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 rule. Since the parity of the magnetic dipole operator is even, then this operator accounts for allowed 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 having a bearing on the energy spectrum and the intra-configuration wavefunctions themselves. Configuration interaction may also be used to 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 [1] 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

[1] 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.

form related to the overlap between the mixed parity parts of the corrected wavefunctions, but they can also be integrated as additional phenomelogical parameters.

Judd-Ofelt theory is based on the intermediate coupling 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]

$$\mathrm{f}_{\text{f-ED}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S'L'J'\rangle\right) = \mathcal{R}\frac{8\pi^2 m_e}{3h}\frac{\nu}{2J+1}\frac{\chi}{n}\sum_{k=2,4,6}\Omega_{(k)}\left|\langle \underline{\mathrm{f}}^n \alpha SLJ\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha' S'L'J'\rangle\right|^2, \tag{75}$$

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

$$\chi = \frac{n^2+2}{3} \tag{76}$$

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

5.2.2 Magnetic dipole transitions

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

$$\mathrm{f}_{\text{MD}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S'L'J'\rangle\right) = \frac{2}{3}\frac{\mathcal{E}(|\alpha' S'L'J'\rangle) - \mathcal{E}(|\alpha LSJ)\rangle}{2J+1}\left|\langle \alpha LSJ\|\frac{1}{2}(\hat{L}+g\hat{S})\|\alpha' S'L'J'\rangle\right|^2 \tag{77}$$

where $\mathcal{E}(|\alpha LSJ\rangle)$ is the energy of level $|\alpha LSJ\rangle$.

In **qlanth** this relationship is implemented in the function `IntermediateOscillatorStrengthMD`.

```
IntermediateOscillatorStrengthMD::usage = "
    IntermediateOscillatorStrengthMD[numE, intermediateParams] uses
    Judd-Ofelt theory to estimate the forced electric dipole
    oscillator strengths for an ion whose intermediate coupling
    description is determined by intermediateParams.
The function returns a square array, oStrengthArray, where
    oStrengthArray[[i,j]] equals the oscillator strength (which is a
    dimensionless quantity) between levels |Subscript[\[Psi], i]> and
    |Subscript[\[Psi], j]>.
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 intermediate coupling.
  - levelLabels : A list with the labels of the major components of
    the intermediate coupling levels.
  - magDipoleOstrength : A square array whose elements represent the
    magnetic dipole oscillator strengths between the levels of the
    intermediate coupling eigenstates 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 function admits the following two 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.
";
Options[IntermediateOscillatorStrengthMD] = {
  "PrintFun" -> PrintTemporary
};
IntermediateOscillatorStrengthMD[numE_,
    intermediateParams_Association,  OptionsPattern[]] := Module[
  {
    PrintFun, eigenSys, eigenEnergies, interLevels, energyDiffs,
    levelJs, magDipoleOstrength, LSJmultiplets, majorComponentIndices,
     levelLabels
  },
  (
    PrintFun = OptionValue["PrintFun"];
    PrintFun["> Calculating the intermediate levels for the given
    parameters ..."];
    {basis, eigenSys} = IntermediateSolver[numE, intermediateParams];
    (* The change of basis matrix to the eigenstate basis *)
    eigenEnergies  = First /@ eigenSys;
    interLevels    = Transpose[Last /@ eigenSys];
    energyDiffs    = Abs@Outer[Subtract, eigenEnergies, eigenEnergies
    ];
    energyDiffs   *= kayserToHartree;
    levelJs        = #[[2]] & /@ eigenSys;
    magDip         = IntermediateMagDipole[numE];
    magDipoleOstrength = (2/3 *
      αFine^2
      energyDiffs *
      Abs[Transpose[interLevels] . magDip . interLevels]^2
    );
    magDipoleOstrength = MapIndexed[
      1/(2 * levelJs[[#2[[1]]]] + 1) * #1 &,
      magDipoleOstrength,
      {2}
    ];
    LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
    InputForm[#[[2]]]]) & /@ basis;
    majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ Transpose[
    interLevels];
    levelLabels = LSJmultiplets[[majorComponentIndices]];
    Return[{basis, eigenSys, levelLabels, magDipoleOstrength}];
  )
];
```

6 Data fitting

qlanth also has the capacity to fit the Hamiltonian to experimental data. This is included in the sub-module `fittings.m`.

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.

It requires the following parameters:

- `numE`: number of electrons in the system, specifying the electronic configuration.
- `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 Kramer's 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 experiemental data which is present, but which is considered dubious. In the case of odd electron configurations these indices need to implicitly include the double degeneracy of Kramer's 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 Carnall's systematic study of LaF_3.
- `startValues`: an assocciation 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 contraints 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. **Intermediate Coupling Basis Calculation**: Computes an intermediate coupling basis using free-ion parameters, aiding in the energy level analysis of the system.

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.

- `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 whith 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 basis.
- `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 define the intermediate coupling basis.

- `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`: Aa 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.

Table 𝓕ig-2 shows the result of fitting the experimental data included in Carnall, in which certain parameters are held fixed, others made proportional to one another, and the other fitted through the Levenberg-Marquard method.

	Ce	Pr	Nd	Pm	Sm	Eu	Gd	Tb	Dy	Ho	Er	Tm	Yb
F2	---	68860.±20.	73020.±10.	[76400.]	79700.±30.	83080.±30.	85640.±10.	88870.±20.	91830.±40.	94560.±30.	97570.±20.	100130.±20.	---
F4	---	50400.±70.	52770.±40.	[54900.]	57260.±30.	[59240.2]	[60809.]	[62834.6]	64350.±30.	66480.±40.	68050.±40.	69660.±90.	---
F6	---	32880.±60.	35750.±50.	[37700.]	40200.±20.	[42539.9]	44790.±10.	47190.±10.	49260.±20.	51900.±50.	54180.±60.	56030.±80.	---
ζ	647.±1.	749.±1.	885.1±0.8	[1025.]	1175.±1.	1332.±2.	1509.±3.	1705.±2.	1908.±1.	2139.±1.	2377.±1.	2634.±1.	2915.±1.
α	---	16.1±0.2	21.3±0.1	[20.5]	20.3±0.1	[20.16]	18.8±0.1	18.5±0.1	18.2±0.1	17.7±0.1	17.4±0.1	16.9±0.2	---
β	---	−550.±10.	−580.±10.	[−560.]	−572.±5.	[−566.9]	[−600.]	−586.±4.	−638.±6.	−615.±8.	−580.±10.	−610.±10.	---
γ	---	1360.±10.	1430.±10.	[1475.]	[1500.]	[1500.]	[1575.]	[1650.]	1802.±5.	[1800.]	[1800.]	[1820.]	---
T2	---	---	293.±4.	[300.]	[300.]	[300.]	[300.]	[320.]	315.±5.	[400.]	[400.]	[400.]	---
T3	---	---	35.±9.	[35.]	[36.]	[40.]	[42.]	[40.]	30.±10.	36.±8.	40.±10.	---	---
T4	---	---	59.±8.	[58.]	[56.]	[60.]	[62.]	[50.]	90.±40.	96.±7.	63.±9.	---	---
T6	---	---	−280.±20.	[−310.]	−330.±30.	[−300.]	[−295.]	−350.±40.	−290.±40.	−260.±50.	−280.±20.	---	---
T7	---	---	330.±20.	[350.]	360.±20.	[370.]	[350.]	320.±30.	370.±20.	300.±40.	330.±20.	---	---
T8	---	---	300.±20.	[320.]	340.±10.	[320.]	[310.]	330.±10.	320.±20.	340.±20.	360.±20.	---	---
M0	---	1.8±0.3	2.1±0.1	[2.4]	2.52±0.06	[2.1]	3.3±0.1	2.4±0.09	3.22±0.06	2.61±0.08	3.8±0.1	3.9±0.2	---
P2	---	−30.±30.	210.±10.	[275.]	330.±10.	[360.]	720.±40.	390.±20.	620.±10.	550.±20.	680.±20.	670.±40.	---
B02	[−218.]	−220.±20.	−250.±40.	[−245.]	−210.±30.	−210.±60.	[−231.]	−240.±40.	−230.±20.	[−240.]	−230.±30.	−250.±30.	[−249.]
B04	[738.]	730.±30.	500.±100.	[470.]	300.±200.	400.±100.	[604.]	600.±100.	560.±70.	500.±100.	300.±100.	450.±60.	[457.]
B06	[679.]	670.±40.	640.±40.	[640.]	600.±100.	500.±100.	[280.]	300.±100.	170.±90.	300.±100.	440.±70.	300.±60.	[282.]
B22	[−50.]	−120.±20.	−50.±30.	[−50.]	[−50.]	[−50.]	[−99.]	−100.±50.	−60.±10.	−100.±20.	−90.±30.	−100.±20.	[−105.]
B24	[431.]	420.±50.	500.±80.	[525.]	620.±50.	[597.]	[340.]	260.±70.	190.±70.	240.±60.	350.±90.	300.±40.	[320.]
B26	[−921.]	−910.±50.	−830.±40.	[−750.]	−680.±90.	[−706.]	[−721.]	−730.±80.	−670.±50.	−550.±60.	−480.±20.	−450.±20.	[−482.]
B44	[616.]	600.±30.	570.±50.	[490.]	430.±60.	[408.]	[452.]	480.±30.	550.±30.	460.±40.	300.±100.	430.±40.	[428.]
B46	[−348.]	−350.±20.	−400.±40.	[−450.]	−400.±100.	[−508.]	[−204.]	−240.±80.	−100.±100.	−200.±30.	−230.±30.	−240.±60.	[−234.]
B66	[−788.]	−780.±60.	−830.±30.	[−760.]	−730.±80.	[−692.]	[−509.]	−520.±90.	−540.±40.	−580.±30.	−500.±20.	−500.±30.	[−492.]
ϵ	−2.9	−2.1	−4.8	−8.2	−16.4	−12.5	20.8	−6.	−6.7	−4.9	−7.3	−10.4	−32.8
σ	47	16	13	4	13	17	10	9	11	9	14	11	61
σBill	51	16	14	0	13	16	10	12	12	10	19	10	38
n	7	75	146	284	233	29	70	146	198	204	127	56	5
nBill	7	75	146	0	232	29	70	146	198	204	127	56	5

Figure 2: Fitting the data from Carnall et. al using **`qlanth`**

```
ClassicalFit::usage="Classical[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.
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.
21  \"paramSols\" which is a list of the parameters during the stepping
        of the fitting algorithm.
22  \"timeTaken/s\" which is the time taken to find the best fit.
23  \"simplifier\" which is the simplifier used to simplify the
        Hamiltonian.
24  \"excludeDataIndices\" as given in the input.
25  \"starValues\" as given in the input.
26
27  \"freeIonSymbols\" which are the symbols used in the intermediate
        coupling basis.
28  \"truncationEnergy\" which is the energy used to truncate the
        Hamiltonian.
29  \"numE\" which is the number of electrons in the f^numE configuration
        .
30  \"expData\" which is the experimental data used for fitting.
31  \"problemVars\" which are the symbols considered for fitting
32
33  \"maxIterations\" which is the maximum number of iterations used by
        NMinimize.
34  \"hamDim\" which is the dimension of the full Hamiltonian.
35  \"allVars\" which are all the symbols defining the Hamiltonian under
        the aggregate simplifications.
36  \"freeBies\" which are the free-ion parameters used to define the
        intermediate coupling basis.
37  \"truncatedDim\" which is the dimension of the truncated Hamiltonian.
38  \"compiledIntermediateFname\" the file name of the compiled function
        used for the truncated Hamiltonian.
39
40  \"fittedLevels\" which is the number of levels fitted for.
41  \"actualSteps\" the number of steps that FindMiniminum actually took.
42  \"solWithUncertainty\" which is a list of replacement rules whose
        left hand sides are symbols for the used parameters and whose's
        right hand sides are lists with the best fit value and the
        uncertainty in that value.
43  \"rmsHistory\" which is a list of the \[Sigma] values found during
        the fitting.
44  \"Appendix\" which is an association appended to the log file under
        the key \"Appendix\".
45  \"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.
46
47  \"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.
```

```
48 \"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.
49
50 The function admits the following options with default values:
51   \"MaxHistory\" : determines how long the logs for the solver can be
       .
52   \"MaxIterations\": determines the maximum number of iterations used
        by NMinimize.
53   \"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\".
54   \"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.
55
56   \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
     5.
57   \"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.
58   \"PrintFun\": the function used to print progress messages, the
     default is PrintTemporary.
59
60   \"SlackChannel\": name of the Slack channel to which to dump
     progress messaages, the default is None which disables this option
       .
61   \"ProgressView\": whether or not a progress window will be opened
     to show the progress of the solver, the default is True.
62   \"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
      startingValues.
63   \"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.
64
65   \"AppendToLogFile\": an association appended to the log file under
     the key \"Appendix\".
66   \"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.
67   \"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.
68
69   \"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,
  "SlackChannel"     -> None,
  "ProgressView"     -> True,
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "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, T11->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->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, T11->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, T11->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, T11->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, \[Sigma]exp_?NumericQ,
    constraints_List, OptionsPattern[]]:=
  (* Module[{accuracyGoal, activeVarIndices, activeVars,
    activeVarsString, activeVarsWithRange, allFreeEnergies,
    allFreeEnergiesSorted, allVars, allVarsVec,
    argsForEvalInsideOfTheIntermediateSystems,
    argsOfTheIntermediateEigensystems, aVar, aVarPosition, basis,
    basisChanger, basisChangerBlocks, bestError, bestParams, bestRMS,
    blockShifts, blockSizes, colIdx, compiledDiagonal,
    compiledIntermediateFname, constrainedProblemVars,
    constrainedProblemVarsList, covMat, currentRMS, degressOfFreedom,
    dependentVars, diagonalBlocks, diagonalScalarBlocks, diff,
    eigenEnergies, eigenvalueDispenserTemplate, eigenVectors,
    elevatedIntermediateEigensystems, endTime, fmSol, fmSolAssoc,
    fractionalWidth, freeBies, freeIenergiesAndMultiplets,
    freeionSymbols, fullHam, fullSolVec, funcString, ham, hamDim,
    hamEigenvaluesTemplate, hamString, hess, indepSolVecVec, indepVars
    , intermediateHam, isolationValues, jobVars, lin, linMat, ln,
    lnParams, logFilePrefix, logFname, magneticSimplifier,
    maxFreeEnergy, maxHistory, maxIterations, methodString,
    methodStringTemplate, minFreeEnergy, minpoly, modelSymbols,
    multipletAssignments, needlePosition, numBlocks, numQSignature,
    numReps, solCompendium, openNotebooks, ordering, othersFixed,
    otherSimplifier, p0, paramBest, paramSigma, perHam, polySols,
    presentDataIndices, PrintFun, problemVarsPositions, problemVarsQ,
    problemVarsQString, problemVarsVec, problemVarsWithStartValues,
    reducedModelSymbols, resultMessage, roundedTruncationEnergy,
    rowIdx, runningInteractive, shiftToggle, simplifier, slackChan,
    sol, solAssoc, sols, solWithUncertainty, sortedTruncationIndex,
    sqdiff, standardValues, starTime, startingValues, startTime,
```

```
    startVarValues, states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, thisPoly, threadHeaderTemplate
    , threadMessage, threadTS, timeTaken, totalVariance,
    truncadedFname, truncatedIntermediateBasis,
    truncatedIntermediateHam, truncationEnergy, truncationIndices,
    truncationUmbral, usingInitialRange, varHash, varIdx,
    varsWithConstants, varWithValsSignature, \[Lambda]0Vec, \[Lambda]
    exp}, *)
143 Module[{},
144 (
145   solCompendium   = <||>;
146   addShift      = OptionValue["AddConstantShift"];
147   ln            = theLanthanides[[numE]];
148   maxHistory    = OptionValue["MaxHistory"];
149   maxIterations = OptionValue["MaxIterations"];
150   logFilePrefix = If[OptionValue["FilePrefix"] == "",
151                      ToString[theLanthanides[[numE]]],
152                      OptionValue["FilePrefix"]
153                    ];
154   accuracyGoal = OptionValue["AccuracyGoal"];
155   slackChan    = OptionValue["SlackChannel"];
156   PrintFun     = OptionValue["PrintFun"];
157   freeIonSymbols      = OptionValue["FreeIonSymbols"];
158   runningInteractive  = (Head[$ParentLink] === LinkObject);
159   magneticSimplifier  = OptionValue["MagneticSimplifier"];
160   magFieldSimplifier  = OptionValue["MagFieldSimplifier"];
161   symmetrySimplifier  = OptionValue["SymmetrySimplifier"];
162   otherSimplifier     = OptionValue["OtherSimplifier"];
163   threeBodySimplifier = If[Head[OptionValue["ThreeBodySimplifier"]]
    == Association,
164                            OptionValue["ThreeBodySimplifier"][numE],
165                            OptionValue["ThreeBodySimplifier"]
166                          ];
167
168   truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic
    ,
169     PrintFun["Truncation energy set to Automatic, using the maximum
     energy in the data ..."];
170     Max[Select[First /@ expData, NumericQ[#] &]],
171     OptionValue["TruncationEnergy"]
172     ];
173   PrintFun["Using a truncation energy of ", truncationEnergy, " K"
    ];
174
175   simplifier          = Join[magneticSimplifier,
176                              magFieldSimplifier,
177                              symmetrySimplifier,
178                              threeBodySimplifier,
179                              otherSimplifier];
180
181   PrintFun["Determining gaps in the data ..."];
182   (* the indices that are numeric in expData whatever is non-
    numeric is assumed as a known gap *)
183   presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
```

```
    , ___}]];
184 (* some indices omitted here based on the excludeDataIndices
    argument *)
185 presentDataIndices = Complement[presentDataIndices,
    excludeDataIndices];
186
187 hamDim                              = Binomial[14, numE];
188 solCompendium["simplifier"]          = simplifier;
189 solCompendium["excludeDataIndices"] = excludeDataIndices;
190 solCompendium["startValues"]         = startValues;
191 solCompendium["freeIonSymbols"]      = freeIonSymbols;
192 solCompendium["truncationEnergy"]    = truncationEnergy;
193 solCompendium["numE"]                = numE;
194 solCompendium["expData"]             = expData;
195 solCompendium["problemVars"]         = problemVars;
196 solCompendium["maxIterations"]       = maxIterations;
197 solCompendium["hamDim"]              = hamDim;
198 solCompendium["constraints"]         = constraints;
199 modelSymbols  = Sort[Select[paramSymbols, Not[MemberQ[Join[
    racahSymbols, chenSymbols,{t2Switch, \[Epsilon],gs}],#]]&]];
200 (* remove the symbols that will be removed by the simplifier, no
    symbol should remain here that is not in the symbolic Hamiltonian
    *)
201 reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
    simplifier],#]]&];
202
203 (* this is useful to understand what are the arguments of the
    truncated compiled Hamiltonian *)
204 If[OptionValue["SignatureCheck"],
205   (
206     Print["Given the model parameters and the simplifying
    assumptions, the resultant model parameters are:"];
207     Print[{reducedModelSymbols}];
208     Print["Exiting ..."];
209     Return[""];
210   )
211 ];
212
213 (* calculate the basis *)
214 PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
215 basis    = BasisLSJMJ[numE];
216
217 (* get the reference parameters from LaF3 *)
218 PrintFun["Getting reference free-ion parameters for ",ln," using
    LaF3 ..."];
219 lnParams = LoadParameters[ln];
220 freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
      numE];
221 (* a more explicit alias *)
222 allVars           = reducedModelSymbols;
223 numericConstraints = Association@Select[constraints, NumericQ
    [#[[2]]] &];
224 standardValues    = allVars /. Join[lnParams, numericConstraints
    ];
225 solCompendium["allVars"]  = allVars;
```

```
226     solCompendium["freeBies"] = freeBies;
227
228     (* reload compiled version if found *)
229     varHash                   = Hash[{numE, allVars, freeBies,
        truncationEnergy, simplifier}];
230     compiledIntermediateFname = ln<>"-compiled-intermediate-truncated
        -ham-"<>ToString[varHash]<>".mx";
231     compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
        compiledIntermediateFname}];
232     solCompendium["compiledIntermediateFname"] =
        compiledIntermediateFname;
233
234     If[FileExistsQ[compiledIntermediateFname],
235       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 ..."];
236       PrintFun["Using : ", compiledIntermediateFname];
237       {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
         Import[compiledIntermediateFname];,
238     (
239       (* grab the Hamiltonian preserving its block structure *)
240       PrintFun["Assembling the Hamiltonian for f^",numE," keeping the
         block structure ..."];
241       ham           = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
242       (* apply the simplifier *)
243       PrintFun["Simplifying using the aggregate set of simplification
         rules ..."];
244       ham           = Map[ReplaceInSparseArray[#, simplifier]&, ham,
        {2}];
245       PrintFun["Zeroing out every symbol in the Hamiltonian that is
        not a free-ion parameter ..."];
246       (* Get the free ion symbols *)
247       freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
         freeIonSymbols];
248       (* Take the diagonal blocks for the intermediate analysis *)
249       PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."
        ];
250       diagonalBlocks       = Diagonal[ham];
251       (* simplify them to only keep the free ion symbols *)
252       PrintFun["Simplifying the diagonal blocks to only keep the free
         ion symbols ..."];
253       diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
        ]&/@diagonalBlocks;
254       (* these include the MJ quantum numbers, remove that *)
255       PrintFun["Contracting the basis vectors by removing the MJ
        quantum numbers from the diagonal blocks ..."];
256       diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
257
258       argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
        Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
259       argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
        ToString[#]<>"v") & /@ freeIonSymbols,", "]];
260       PrintFun["argsOfTheIntermediateEigensystems = ",
        argsOfTheIntermediateEigensystems];
```

```
261     PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
    argsForEvalInsideOfTheIntermediateSystems];
262     PrintFun["(if the following fails, it might help to see if the
    arguments of TheIntermediateEigensystems match the ones shown
    above)"];
263
264     (* 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 *)
265     (* compile one function for each of the blocks *)
266     PrintFun["Compiling functions for the diagonal blocks of the
    Hamiltonian ..."];
267     compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N
    [Normal[#]]]]&/@diagonalScalarBlocks;
268     (* use that to create a function that will calculate the free-
    ion eigensystem *)
269     TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζv_
    ] := (
270       theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&)/
    @compiledDiagonal;
271       theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
272       Js      = AllowedJ[numEv];
273       basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
274       (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
275       elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2
    Js[[#2[[1]]]]+1]&, theIntermediateEigensystems];
276       (* Identify a single MJ to keep *)
277       pivot = If[EvenQ[numEv],0,-1/2];
278       LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
279       (* calculate the multiplet assignments that the intermediate
    basis eigenvectors have *)
280       needlePosition = 0;
281       multipletAssignments = Table[
282         (
283           J          = Js[[idx]];
284           eigenVecs = theIntermediateEigensystems[[idx]][[2]];
285           majorComponentIndices         = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
286           majorComponentIndices        += needlePosition;
287           needlePosition               += Length[
    majorComponentIndices];
288           majorComponentAssignments     = LSJmultiplets[[#]]&/
    @majorComponentIndices;
289           (* All of the degenerate eigenvectors belong to the same
    multiplet*)
290           elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments,2J+1];
291           elevatedMultipletAssignments
292         ),
293       {idx, 1, Length[Js]}
294       ];
295       (* put together the multiplet assignments and the energies *)
296       freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
```

```
    @elevatedIntermediateEigensystems, multipletAssignments}];
297      freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
298      (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
299      basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
300      basisChanger = SparseArray[basisChanger];
301      Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
302    );
303
304    PrintFun["Calculating the intermediate eigensystems for ",ln,"
    using free-ion params from LaF3 ..."];
305    (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
306    {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
307
308    (* use that intermediate coupling basis to compile a function
    for the full Hamiltonian *)
309    allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
310    (* important that the intermediate coupling basis have attached
     energies, which make possible the truncation *)
311    ordering = Ordering[allFreeEnergies];
312    (* sort the free ion energies and determine which indices
    should be included in the truncation *)
313    allFreeEnergiesSorted          = Sort[allFreeEnergies];
314    {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
315    (* determine the index at which the energy is equal or larger
    than the truncation energy *)
316    sortedTruncationIndex = Which[
317      truncationEnergy > (maxFreeEnergy-minFreeEnergy),
318      hamDim,
319      True,
320      FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted
    ],x_/;x>truncationEnergy,{0},1][[1]]
321    ];
322    (* the actual energy at which the truncation is made *)
323    roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
324
325    (* the indices that enact the truncation *)
326    truncationIndices = ordering[[;;sortedTruncationIndex]];
327    (* Return[{basisChanger, ham, truncationIndices}]; *)
328    PrintFun["Computing the block structure of the change of basis
    array ..."];
329    blockSizes = BlockArrayDimensionsArray[ham];
330    basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
331    blockShifts = First /@ Diagonal[blockSizes];
332    numBlocks = Length[blockSizes];
333    (* using the ham (with all the symbols) change the basis to the
```

```
 computed one *)
  PrintFun["Changing the basis of the Hamiltonian to the intermediate coupling basis ..."];
  (* intermediateHam            = Transpose[basisChanger].ham.basisChanger; *)
  (* Return[{basisChangerBlocks, ham}]; *)
  intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks];
  PrintFun["Distributing products inside of symbolic matrix elements to keep complexity in check ..."];
  Do[
    intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
    {rowIdx, 1, numBlocks},
    {colIdx, 1, numBlocks}
  ];
  intermediateHam = BlockMatrixMultiply[BlockTranspose[basisChangerBlocks],intermediateHam];
  PrintFun["Distributing products inside of symbolic matrix elements to keep complexity in check ..."];
  Do[
    intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
    {rowIdx, 1, numBlocks},
    {colIdx, 1, numBlocks}
  ];
  (* using the truncation indices truncate that one *)
  PrintFun["Truncating the Hamiltonian ..."];
  truncatedIntermediateHam = TruncateBlockArray[intermediateHam, truncationIndices, blockShifts];
  (* these are the basis vectors for the truncated hamiltonian *)
  PrintFun["Saving the truncated intermediate basis ..."];
  truncatedIntermediateBasis = basisChanger[[All, truncationIndices]];

  PrintFun["Compiling a function for the truncated Hamiltonian ..."];
  (* 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[truncatedIntermediateHam]];
  (* save the compiled function *)
  PrintFun["Saving the compiled function for the truncated Hamiltonian and the truncated intermediate basis ..."];
  Export[compiledIntermediateFname, {compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
 )
 ];

 truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
 PrintFun["The truncated Hamiltonian has a dimension of ", truncationUmbral, "x", truncationUmbral, " ..."];
 presentDataIndices = Select[presentDataIndices, # <= truncationUmbral &];
 solCompendium["presentDataIndices"] = presentDataIndices;
```

```
371
372     (* the problemVars are the symbols that will be fitted for *)
373
374     PrintFun["Starting up the fitting process using the Levenberg-
        Marquardt method ..."];
375     (* using the problemVars I need to create the argument list
        including _?NumericQ *)
376     problemVarsQ = (ToString[#] <> "_?NumericQ") & /@ problemVars;
377     problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
378     (* we also need to have the padded arguments with the variables
        in the right position and the fixed values in the remaining ones
        *)
379     problemVarsPositions = Position[allVars, #][[1, 1]] & /@
        problemVars;
380     problemVarsString = StringJoin[Riffle[ToString /@ problemVars, ",
         "]];
381     (* to feed parameters to the Hamiltonian, which includes all
        parameters, we need to form the rist set of arguments, with fixed
        values where needed, and the variables in the right position *)
382     varsWithConstants = standardValues;
383     varsWithConstants[[problemVarsPositions]] = problemVars;
384     varsWithConstantsString = ToString[varsWithConstants];
385
386     (* this following function serves eigenvalues from the
        Hamiltonian, has memoization so it might grow to use a lot of RAM
        *)
387     Clear[HamSortedEigenvalues];
388     hamEigenvaluesTemplate = StringTemplate["
389     HamSortedEigenvalues[`problemVarsQ`]:=(
390       ham          = compileIntermediateTruncatedHam@@`
        varsWithConstants`;
391       eigenValues = Sort@Eigenvalues@ham;
392       eigenValues = eigenValues - Min[eigenValues];
393       HamSortedEigenvalues[`problemVarsString`] = eigenValues;
394       Return[eigenValues]
395     )"];
396     hamString = hamEigenvaluesTemplate[<|
397         "problemVarsQ" -> problemVarsQString,
398         "varsWithConstants" -> varsWithConstantsString,
399         "problemVarsString" -> problemVarsString
400         |>];
401     ToExpression[hamString];
402
403     (* 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 *)
404     eigenvalueDispenserTemplate = StringTemplate["
405     PartialHamEigenvalues[`problemVarsQ`][i_]:=(
406       eigenVals = HamSortedEigenvalues[`problemVarsString`];
407       eigenVals[[i]]
408     )
409     "];
410     eigenValueDispenserString =
411       eigenvalueDispenserTemplate[<|
412         "problemVarsQ"      -> problemVarsQString,
```

```
413         "problemVarsString" -> problemVarsString
414         |>];
415     ToExpression[eigenValueDispenserString];
416 
417     PringFun["Determining the free variables after constraints ..."];
418     constrainedProblemVars = (problemVars /. constraints);
419     constrainedProblemVarsList = Variables[constrainedProblemVars];
420     If[addShift,
421       PrintFun["Adding a constant shift to the fitting parameters ...
      "];
422       constrainedProblemVarsList = Append[constrainedProblemVarsList,
       \[Epsilon]]
423     ];
424 
425     indepVars = Complement[pVars, #[[1]] & /@ constraints];
426     stringPartialVars = ToString/@constrainedProblemVarsList;
427 
428     paramSols  = {};
429     rmsHistory = {};
430     steps      = 0;
431     problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
432     If[addShift,
433       problemVarsWithStartValues = Append[problemVarsWithStartValues,
       {\[Epsilon] ,0}];
434     ];
435     openNotebooks     = If[runningInteractive,
436               ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
      [],
437               {}];
438     If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
      ProgressView"],
439       ProgressNotebook["Basic"->False]
440     ];
441     degressOfFreedom = Length[presentDataIndices] - Length[
      problemVars] - 1;
442     PrintFun["Fitting for ", Length[presentDataIndices], " data
      points with ", Length[problemVars], " free parameters.", " The
      effective degrees of freedom are ", degressOfFreedom, " ..."];
443 
444     PrintFun["Starting the fitting process ..."];
445     startTime=Now;
446     shiftToggle = If[addShift, 1, 0];
447     sol = FindMinimum[
448       Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@
      constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
449       {j, presentDataIndices}],
450       problemVarsWithStartValues,
451       Method        -> "LevenbergMarquardt",
452       MaxIterations -> OptionValue["MaxIterations"],
453       AccuracyGoal  -> OptionValue["AccuracyGoal"],
454       StepMonitor :> (
455         steps      += 1;
456         currentRMS = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues
      @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j,
      presentDataIndices}];
```

```
457         currentRMS = Sqrt[currentRMS / degressOfFreedom];
458         paramSols  = AddToList[paramSols, constrainedProblemVarsList,
     maxHistory];
459         rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
460         )
461       ];
462     endTime = Now;
463     timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
464     PrintFun["Solution found in ", timeTaken, "s"];
465
466     solVec = constrainedProblemVars /. sol[[-1]];
467     indepSolVec = indepVars /. sol[[-1]];
468     If[addShift,
469       \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
470       \[Epsilon]Best = 0
471     ];
472     fullSolVec = standardValues;
473     fullSolVec[[problemVarsPositions]] = solVec;
474     PrintFun["Calculating the numerical Hamiltonian corresponding to
    the solution ..."];
475     fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
476     PrintFun["Calculating energies and eigenvectors ..."];
477     {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
478     states = Transpose[{eigenEnergies, eigenVectors}];
479     states = SortBy[states, First];
480     eigenEnergies = First /@ states;
481     PrintFun["Shifting energies to make ground state zero of energy
    ..."];
482     eigenEnergies = eigenEnergies - eigenEnergies[[1]];
483     PrintFun["Calculating the linear approximant to each eigenvalue
    ..."];
484     allVarsVec = Transpose[{allVars}];
485     p0 = Transpose[{fullSolVec}];
486     linMat = {};
487     If[addShift,
488       tail = -2,
489       tail = -1];
490     Do[
491       (
492         aVarPosition = Position[allVars, aVar][[1, 1]];
493         isolationValues = ConstantArray[0, Length[allVars]];
494         isolationValues[[aVarPosition]] = 1;
495         dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
    constraints]];
496         Do[
497           isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
498           {dVar, dependentVars}
499         ];
500         perHam  = compileIntermediateTruncatedHam @@ isolationValues;
501         lin     = FirstOrderPerturbation[Last /@ states, perHam];
502         linMat  = Append[linMat, lin];
503       ),
504       {aVar, constrainedProblemVarsList[[;;tail]]}
505     ];
```

```
506     PrintFun["Removing the gradient of the ground state ..."];
507     linMat = (# - #[[1]] & /@ linMat);
508     PrintFun["Transposing derivative matrices into columns ..."];
509     linMat  = Transpose[linMat];
510
511     PrintFun["Calculating the eigenvalue vector at solution ..."];
512     \[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
513     PrintFun["Putting together the experimental vector ..."];
514     \[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
        ]]}];
515     problemVarsVec = If[addShift,
516       Transpose[{constrainedProblemVarsList[[;;-2]]}],
517       Transpose[{constrainedProblemVarsList}]
518     ];
519     indepSolVecVec = Transpose[{indepSolVec}];
520     PrintFun["Calculating the difference between eigenvalues at
        solution ..."];
521     diff = If[linMat=={},
522       (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
523       (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
        presentDataIndices]].(problemVarsVec - indepSolVecVec)
524     ];
525     PrintFun["Calculating the sum of squares of differences around
        solution ... "];
526     sqdiff         = Expand[(Transpose[diff] . diff)[[1, 1]]];
527     PrintFun["Calculating the minimum (which should coincide with sol
         ..."];
528     minpoly        = sqdiff /. AssociationThread[problemVars -> solVec
        ];
529     fmSolAssoc     = Association[sol[[2]]];
530     totalVariance = Length[presentDataIndices]*\[Sigma]exp^2;
531     PrintFun["Calculating the uncertainty in the parameters ..."];
532     solWithUncertainty = Table[
533       (
534         aVar         = constrainedProblemVarsList[[varIdx]];
535         paramBest    = aVar /. fmSolAssoc;
536         othersFixed = AssociationThread[Delete[
        constrainedProblemVarsList[[;;tail]], varIdx] -> Delete[
        indepSolVec, varIdx]];
537         thisPoly     = sqdiff /. othersFixed;
538         polySols     = Last /@ Last /@ Solve[thisPoly == minpoly + 1*
        totalVariance];
539         polySols     = Select[polySols, Im[#] == 0 &];
540         paramSigma   = Max[polySols] - Min[polySols];
541         (aVar -> {paramBest, paramSigma})
542       ),
543     {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
544     ];
545     PrintFun["Calculating the covariance matrix ..."];
546     hess = If[linmat=={},
547       {{Infinity}},
548       2 * Transpose[linMat[[presentDataIndices]]] . linMat[[
        presentDataIndices]]
549     ];
550     covMat   = If[linmat=={},
```

```
      {{0}},
      \[Sigma]exp^2 * Inverse[hess]
    ];
    bestRMS = Sqrt[minpoly / degressOfFreedom];
    solCompendium["truncatedDim"] = truncationUmbral;
    solCompendium["fittedLevels"] = Length[presentDataIndices];
    solCompendium["actualSteps"] = steps;
    solCompendium["bestRMS"]     = bestRMS;
    solCompendium["solWithUncertainty"] = solWithUncertainty;
    solCompendium["problemVars"] = problemVars;
    solCompendium["paramSols"]   = paramSols;
    solCompendium["rmsHistory"]  = rmsHistory;
    solCompendium["Appendix"]    = OptionValue["AppendToLogFile"];
    solCompendium["timeTaken/s"] = timeTaken;
    solCompendium["bestParams"]  = sol[[2]];
    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;
      )
    ];
    logFname = LogSol[solCompendium, logFilePrefix];
    Return[solCompendium];
  )
];
```

7 Accompanying notebooks

qlanth is accompanied by the following auxiliary Mathematica notebooks:

- qlanth.nb: gives an overview of the different included functions.
- qlanth - Table Generator.nb: generates the basic tables on which every calculation is based.
- qlanth - JJBlock Calculator.nb: can be used to generate the JJ blocks for the different interactions. The data files produced here are necessary for HamMatrixAssembly to work.
- The Lanthanides in LaF3.nb: runs **qlanth** over the lanthanide ions in $LaF_3$and compares the results against the published values from Carnall. It also calculates magnetic dipole transition rates and oscillator strengths.

8 Additional data

8.1 Carnall et al data on Ln:LaF_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. Additionally the function LoadParameters 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.

```
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}
    ";
```

```
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];
  ];
);
```

```
LoadParameters::usage="LoadParameters[ln] takes a string with the
    symbol the element of a trivalent lanthanide ion and returns model
     parameters for it. It is based on the data for LaF3. If the
    option \"Free Ion\" is set to True then the function sets all
    crystal field parameters to zero. Through the option \"gs\" it
    allows modyfing the electronic gyromagnetic ratio. For
    completeness this function also computes the E parameters using
    the F parameters quoted on Carnall.";
Options[LoadParameters] = {
    "Source"->"Carnall",
    "Free Ion"->False,
    "gs"->2.002319304386,
    "With Uncertainties"->False
    };
LoadParameters[Ln_String, OptionsPattern[]]:= Module[
  {
    source, params, uncertain, uncertainKeys, uncertainRules
  },
  (
    If[Not[ValueQ[Carnall]],
      LoadCarnall[];
    ];
    source = OptionValue["Source"];
    params = Which[source=="Carnall",
            (Association[Carnall["data"][Ln]])
            ];
```

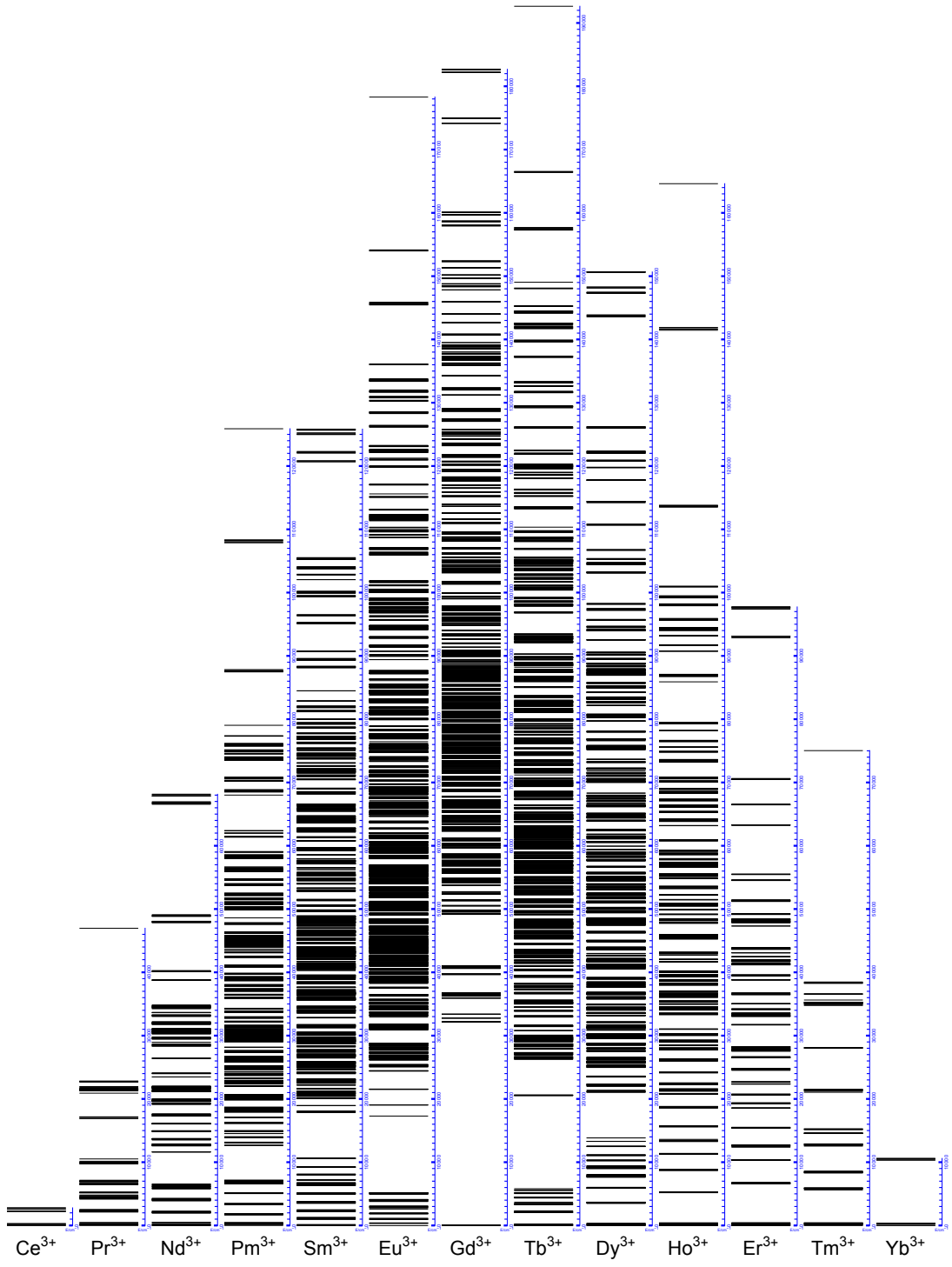

Figure 3: Dieke plot for lanthanum fluoride from data generated by **`qlanth`**.

```
20      (*If a free ion then all the parameters from the crystal field
        are set to zero*)
21      If[OptionValue["Free Ion"],
22        Do[params[cfSymbol] = 0, {cfSymbol, cfSymbols}]
23      ];
24      params[F0] = 0;
25      params[M2] = 0.56*params[M0]; (*See Carnall 1989,Table I,caption,
        probably fixed based on HF values*)
26      params[M4] = 0.31*params[M0]; (*See Carnall 1989,Table I,caption,
        probably fixed based on HF values*)
27      params[P0] = 0;
28      params[P4] = 0.5*params[P2];  (*See Carnall 1989,Table I,caption,
        probably fixed based on HF values*)
29      params[P6] = 0.1*params[P2];  (*See Carnall 1989,Table I,caption,
        probably fixed based on HF values*)
30      params[gs] = OptionValue["gs"];
31      {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[F0
        ], params[F2], params[F4], params[F6]];
32      params[E0] = 0;
33      If[
34        Not[OptionValue["With Uncertainties"]],
35        Return[params],
36        (
37          uncertain     = Association[Carnall["annotations"][Ln]];
38          uncertainKeys = Keys[uncertain];
39          uncertain     = If[# == "Not allowed to vary in fitting." ||
        # == "Interpolated",
40              0., #] & /@ uncertain;
41          paramKeys = Keys[params];
42          uncertainVals = Sort[Intersection[paramKeys, uncertainKeys]]
        /. Association[uncertain];
43          uncertainRules = MapThread[Rule, {Sort[uncertainKeys],
        uncertainVals}];
44          Which[
45            MemberQ[{"Ce", "Yb"}, Ln],
46            (
47              subsetL = {F0};
48              subsetR = {0};
49            ),
50            True,
51            (
52            subsetL = {F0, M2, M4, P0, P4, P6, E0, E1, E2, E3};
53            subsetR = {0, M0*0.65, M0*0.31, 0, P2*0.5, P2*0.1,
54              0,
55              Sqrt[(196 F2^2)/164025 + (49 F4^2)/88209 + (122500 F6^2)
        /134165889],
56              Sqrt[F2^2/4100625 + F4^2/10673289 + (30625 F6^2)
        /2743558264161],
57              Sqrt[F2^2/18225 + (4 F4^2)/1185921 + (30625 F6^2)
        /1803785841]};
58            )
59          ];
60          uncertainRules = Join[uncertainRules, MapThread[Rule, {
        subsetL,subsetR /. uncertainRules}]];
61          uncertainRules = Association[uncertainRules];
```

```
        Which[
          Ln == "Eu",
          (
            uncertainRules[F4] = 12.121;
            uncertainRules[F6] = 15.872;
          ),
          Ln == "Gd",
          (
            uncertainRules[F4] = 12.07;
          ),
          Ln == "Tb",
          (
            uncertainRules[F4] = 41.006;
          )
        ];
        If[MemberQ[{"Eu","Gd","Tb"},Ln],
          (
            uncertainRules[E1] = Sqrt[(196 F2^2)/164025 + (49 F4^2)
    /88209 + (122500 F6^2)/134165889] /. uncertainRules;
            uncertainRules[E2] = Sqrt[F2^2/4100625 + F4^2/10673289 +
    (30625 F6^2)/2743558264161] /. uncertainRules;
            uncertainRules[E3] = Sqrt[F2^2/18225 + (4 F4^2)/1185921 +
     (30625 F6^2)/1803785841] /. uncertainRules;
          )
        ];
        uncertainKeys = First /@ Normal[uncertainRules];
        fullParams = Association[MapThread[Rule, {uncertainKeys,
    MapThread[Around, {uncertainKeys /. params, uncertainKeys /.
    uncertainRules}]}]];
        Return[Join[params, fullParams]]
      )
      ];
  )
];
```

8.2 sparsefn.py

qlanth is also accompanied by seven Python scripts sparsef[1-7].py. Each of these contains a single function effective_hamiltonian_f[1-7] which returns a sparse array 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-71 and setting t2Switch to 0. For configurations at or below $\underline{f}^7$ it is necessary to set t2Switch to 1.

9 Units

All of the matrix elements of the Hamiltonian are calculated using the Kayser (K $\equiv$ 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`"];

(* Physical Constants*)
bohrRadius = 5.29177210903 * 10^-9;
ee         = 1.602176634 * 10^-19;

(* 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"

(* 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 *)

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

(* 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;
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;
```

```
42
43 EndPackage[];
```

10 Notation

orbital angular momentum operator of a single electron

$$\overbrace{\hat{l}} \tag{78}$$

total orbital angular momentum operator

$$\overbrace{\hat{L}} \tag{79}$$

spin angular momentum operator of a single electron

$$\overbrace{\hat{s}} \tag{80}$$

total spin angular momentum operator

$$\overbrace{\hat{S}} \tag{81}$$

Shorthand for all other quantum numbers

$$\overbrace{\Lambda} \tag{82}$$

orbital angular momentum number

$$\overbrace{\underline{\ell}} \tag{83}$$

spinning angular momentum number

$$\overbrace{\underline{s}} \tag{84}$$

Coulomb non-central potential

$$\overbrace{\hat{\mathfrak{C}}} \tag{85}$$

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

$$\overbrace{\langle \Lambda LS \| \hat{O} \| \Lambda' L' S' \rangle} \tag{86}$$

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

$$\overbrace{\langle \Lambda LSJ \| \hat{O} \| \Lambda' L' S' J' \rangle} \tag{87}$$

Spectroscopic term αLS in Russel-Saunders notation

$$\overbrace{{}^{2S+1}\alpha \mathrm{L} \equiv |\alpha LS\rangle} \tag{88}$$

spherical tensor operator of rank k

$$\overbrace{\hat{X}^{(k)}} \tag{89}$$

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

$$\overbrace{\hat{X}^{(k)}_q} \tag{90}$$

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

$$\overbrace{\left(\underline{\ell}^{n-1} \alpha' L' S' \middle\} \underline{\ell}^n \alpha LS\right)} \tag{91}$$

11 Definitions

$$\overbrace{[\![x]\!]}^{\text{two plus one}} \coloneqq 2x+1 \tag{92}$$

$$\overbrace{\hat{u}^{(k)}}^{\text{irreducible unit tensor operator of rank k}} \tag{93}$$

$$\overbrace{\hat{U}^{(k)} \coloneqq \sum_{i=1}^{n} \hat{u}^{(k)}}^{\text{symmetric unit tensor operator for n equivalent electrons}} \tag{94}$$

$$\overbrace{\mathcal{C}_q^{(k)} \coloneqq \sqrt{\frac{4\pi}{2k+1}}\mathrm{Y}_q^{(k)}}^{\text{Renormalized spherical harmonics}} \tag{95}$$

$$\overbrace{\triangle(j_1,j_2,j_3) \coloneqq \begin{cases} 1 & \text{if } j_1 = (j_2+j_3),(j_2+j_3-1),\ldots,|j_2-j_3| \\ 0 & \text{otherwise} \end{cases}}^{\text{Triangle ``delta'' between } j_1,j_2,j_3} \tag{96}$$

12 code

12.1 qlanth.m

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

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

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 all 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 fluoride.

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 Litz n, 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 Kr mer. "50th Anniversary of the J u d d Ofelt Theory: An Experimentalist's View of
the Formalism and Its Application." Journal of Luminescence 136 (2013): 221 39 .
------------------------------------------------------------------ *)

BeginPackage["qlanth`"];
Needs["qonstants`"];
Needs["qplotter`"];
Needs["misc`"];

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

T11:  three-body effective operator parameter T^11
T11p: three-body effective operator parameter T^11'
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)
";
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];
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, T11, 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};
paramFamilies         = Hold[{racahSymbols, chenSymbols,
    slaterSymbols, controlSymbols, cfSymbols, TSymbols,
    pseudoMagneticSymbols, marvinSymbols, casimirSymbols,
    magneticSymbols}];
ReleaseHold[Protect /@ paramFamilies];

(* 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;
BasisLSJ;
BasisLSJMJ;
Bqk;
CFP;
CFPAssoc;

CFPTable;
CFPTerms;
Carnall;
CasimirG2;
CasimirSO3;
CasimirSO7;

Cqk;
CrystalField;
Dk;
ElectrostaticConfigInteraction;
Electrostatic;

ElectrostaticTable;
EnergyLevelDiagram;
EnergyStates;
ExportMZip;
BasisTableGenerator;
EigenLever;
EtoF;
ExportmZip;
fsubk;
fsupk;

FindNKLSTerm;
FindSL;

FreeHam;
FtoE;
GG2U;
GSO7W;
GenerateCFP;
GenerateCFPAssoc;
```

```
304
305 GenerateCFPTable;
306 GenerateCrystalFieldTable;
307 GenerateElectrostaticTable;
308 GenerateReducedUkTable;
309 GenerateReducedV1kTable;
310
311 GenerateSOOandECSOLSTable;
312 GenerateSOOandECSOTable;
313 GenerateSpinOrbitTable;
314 GenerateSpinSpinTable;
315 GenerateT22Table;
316
317 GenerateThreeBodyTables;
318 GenerateThreeBodyTables;
319 Generator;
320 GroundMDOscillatorStrength;
321 HamMatrixAssembly;
322 HamiltonianForm;
323
324 HamiltonianMatrixPlot;
325 HoleElectronConjugation;
326 IntermediateMagDipole;
327 IntermediateSolver;
328 IntermediateOscillatorStrengthED;
329 IntermediateOscillatorStrengthMD;
330 IonSolver;
331 ImportMZip;
332 JJBlockMatrix;
333 JJBlockMagDip;
334 JJBlockMagDipIntermediate;
335 JJBlockMatrixFileName;
336
337 JJBlockMatrixTable;
338 JuddOfeltUkSquared;
339 LabeledGrid;
340 ListRepeater;
341 LoadAll;
342 LoadCFP;
343 LoadCarnall;
344
345 LoadChenDeltas;
346 LoadElectrostatic;
347 LoadGuillotParameters;
348 LoadParameters;
349 LoadSOOandECSO;
350
351 LoadSOOandECSOLS;
352 LoadSpinOrbit;
353 LoadSpinSpin;
354 LoadSymbolicHamiltonians;
355 LoadT11;
356
357 LoadT22;
358 LoadTermLabels;
```

```
LoadThreeBody;
LoadUk;
LoadV1k;

MagneticInteractions;
MagDipoleMatrixAssembly;
MagDipLineStrength;
MapToSparseArray;
MaxJ;
MinJ;
NKCFPPhase;

ParamPad;
ParseStates;
ParseStatesByNumBasisVecs;
ParseStatesByProbabilitySum;
ParseTermLabels;

Phaser;
PrettySaunders;
PrettySaundersSLJ;
PrettySaundersSLJmJ;
PrintL;

PrintSLJ;
PrintSLJM;
ReducedSOOandECSOinf2;
ReducedSOOandECSOinfn;
ReducedT11inf2;

ReducedT22inf2;
ReducedUk;
ReducedUkTable;
ReducedV1kTable;
Reducedt11inf2;

ReplaceInSparseArray;
SimplerSymbolicHamMatrix;
SimplerSymbolicIntermediateHamMatrix;
SOOandECSO;
SOOandECSOTable;
Seniority;

ShiftedLevels;
SixJay;
SpinOrbit;
SpinOrbitTable;
SpinSpin;
SpinSpinTable;

Sqk;
SquarePrimeToNormal;
ReducedT22infn;
TPO;

```

```
TabulateJJBlockMatrixTable;
TabulateJJBlockMagDipTable;
TabulateManyJJBlockMatrixTables;
TabulateManyJJBlockMagDipTables;
ScalarOperatorProduct;
ThreeBodyTable;

ThreeBodyTables;
ThreeJay;
TotalCFIters;
MagDipoleRates;
chenDeltas;
fK;

fnTermLabels;
moduleDir;
symbolicHamiltonians;

(* this selects the function that is applied to calculated matrix
    elements which helps keep down the complexity of the resulting
    expressions *)
SimplifyFun = Expand;

Begin["`Private`"]

  moduleDir = DirectoryName[$InputFileName];
  frontEndAvailable = (Head[$FrontEnd] === FrontEndObject);

  (* ############################################################ *)
  (* ########################## MISC ############################ *)

  TPO::usage = "TPO[x, y, ...] gives the product of 2x+1, 2y+1, ...";
  TPO[args__] := Times @@ ((2*# + 1) & /@ {args});

  Phaser::usage = "Phaser[x] gives (-1)^x.";
  Phaser[exponent_] := ((-1)^exponent);

  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.
    ";
461 SquarePrimeToNormal[squarePrime_] :=
462 (
463   radical = Product[Prime[idx1 - 1] ^ Part[squarePrime, idx1], {
    idx1, 2, Length[squarePrime]}];
464   radical = radical /. {"A" -> 10, "B" -> 11, "C" -> 12, "D" ->
    13};
465   val = squarePrime[[1]] * Sqrt[radical];
466   Return[val];
467 );
468
469 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.
470 The function additionally takes an option \"Print\" that if set to
    True, will print the symbols that were not present in the given
    association.";
471 Options[ParamPad] = {"Print" -> True}
472 ParamPad[params_, OptionsPattern[]] := (
473   notPresentSymbols = Complement[paramSymbols, Keys[params]];
474   If[OptionValue["Print"],
475     Print["Following symbols were not given and are being set to 0:
    ",
476     notPresentSymbols]
477   ];
478   newParams = Transpose[{paramSymbols, ConstantArray[0, Length[
    paramSymbols]]}];
479   newParams = (#[[1]] -> #[[2]]) & /@ newParams;
480   newParams = Association[newParams];
481   newParams = Join[newParams, params];
482   Return[newParams];
483   )
484
485 (* ############################################################ *)
486 (* ###################### Racah Algebra ###################### *)
487
488 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.";
489 ReducedUk[numE_, l_, SL_, SpLp_, k_]:=Module[
490   {spin, orbital, Uk, S, L, Sp, Lp, Sb, Lb, parentSL, cfpSL,
    cfpSpLp, Ukval, SLparents, SLpparents, commonParents, phase},
491     {spin, orbital} = {1/2, 3};
492     {S, L}          = FindSL[SL];
493     {Sp, Lp}        = FindSL[SpLp];
494     If[Not[S == Sp],
495       Return[0]
496     ];
497     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];
  ]

  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}]
     ];
```

```
545     ThreeJay[{j1, m1}, {j2, m2}, {j3, m3}] = threejval);
546
547   ReducedV1k::usage = "ReducedV1k[n, l, SL, SpLp, k] gives the
        reduced matrix element of the spherical tensor operator V^(1k).
        See equation 2-101 in Wybourne 1965.";
548   ReducedV1k[numE_, SL_, SpLp_, k_]:=Module[
549     {Vk1, S, L, Sp, Lp, Sb, Lb, spin, orbital, cfpSL, cfpSpLp,
550     SLparents, SpLpparents, commonParents, prefactor},
551     (
552       {spin, orbital} = {1/2, 3};
553       {S, L}          = FindSL[SL];
554       {Sp, Lp}        = FindSL[SpLp];
555       cfpSL           = CFP[{numE, SL}];
556       cfpSpLp         = CFP[{numE, SpLp}];
557       SLparents       = First /@ Rest[cfpSL];
558       SpLpparents     = First /@ Rest[cfpSpLp];
559       commonParents = Intersection[SLparents, SpLpparents];
560       Vk1 = Sum[(
561           {Sb, Lb} = FindSL[\[Psi]b];
562           Phaser[(Sb + Lb + S + L + orbital + k - spin)] *
563           CFPAssoc[{numE, SL, \[Psi]b}] *
564           CFPAssoc[{numE, SpLp, \[Psi]b}] *
565           SixJay[{S, Sp, 1}, {spin, spin, Sb}] *
566           SixJay[{L, Lp, k}, {orbital, orbital, Lb}]
567         ),
568       {\[Psi]b, commonParents}
569       ];
570       prefactor = numE * Sqrt[spin * (spin + 1) * TPO[spin, S, L, Sp,
        Lp] ];
571       Return[prefactor * Vk1];
572     )
573   ];
574
575   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.";
576   Options[GenerateReducedUkTable] = {"Export" -> True, "Progress" ->
        True};
577   GenerateReducedUkTable[numEmax_Integer:7, OptionsPattern[]]:= (
578     numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
        AllowedNKSLTerms[#]]&/@Range[1, numEmax]] * 4;
579     Print["Calculating " <> ToString[numValues] <> " values for Uk k
        =0,2,4,6."];
580     counter = 1;
581     If[And[OptionValue["Progress"], frontEndAvailable],
582     progBar = PrintTemporary[
583         Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
584           counter}]]]
585       ];
586     ReducedUkTable = Table[
587       (
588         counter = counter+1;
589         {numE, 3, SL, SpLp, k} -> SimplifyFun[ReducedUk[numE, 3, SL,
```

```
    SpLp, k]]
      ),
      {numE, 1, numEmax},
      {SL,   AllowedNKSLTerms[numE]},
      {SpLp, AllowedNKSLTerms[numE]},
      {k, {0, 2, 4, 6}}
    ];
    ReducedUkTable = Association[Flatten[ReducedUkTable]];
    ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
    ReducedUkTable.m"}];
    If[And[OptionValue["Progress"], frontEndAvailable],
      NotebookDelete[progBar]
    ];
    If[OptionValue["Export"],
      (
        Print["Exporting to file " <> ToString[ReducedUkTableFname]];
        Export[ReducedUkTableFname, ReducedUkTable];
      )
    ];
    Return[ReducedUkTable];
  )

  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."
  Options[GenerateReducedV1kTable] = {"Export" -> True, "Progress" ->
     True};
  GenerateReducedV1kTable[numEmax_Integer:7, OptionsPattern[]]:= (
    numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
    AllowedNKSLTerms[#]]&/@Range[1, numEmax]];
    Print["Calculating " <> ToString[numValues] <> " values for Vk1."
    ];
    counter = 1;
    If[And[OptionValue["Progress"], frontEndAvailable],
    progBar = PrintTemporary[
        Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
          counter}]]]
      ];
    ReducedV1kTable = Table[
      (
        counter = counter+1;
        {n, SL, SpLp, 1} -> SimplifyFun[ReducedV1k[n, SL, SpLp, 1]]
      ),
      {n, 1, numEmax},
      {SL, AllowedNKSLTerms[n]},
      {SpLp, AllowedNKSLTerms[n]}
    ];
    ReducedV1kTable = Association[ReducedV1kTable];
    If[And[OptionValue["Progress"], frontEndAvailable],
      NotebookDelete[progBar]
    ];
    exportFname = FileNameJoin[{moduleDir, "data", "ReducedV1kTable.m
    "}];
```

```
635     If[OptionValue["Export"],
636       (
637         Print["Exporting to file "<>ToString[exportFname]];
638         Export[exportFname, ReducedV1kTable];
639       )
640     ];
641     Return[ReducedV1kTable];
642   )
643
644   (* ###################### Racah Algebra ###################### *)
645   (* ########################################################### *)
646
647   (* ########################################################### *)
648   (* ###################### Electrostatic ###################### *)
649
650   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.";
651   fsubk[numE_, orbital_, NKSL_, NKSLp_, k_]:=Module[
652     {terms, S, L, Sp, Lp, termsWithSameSpin, SL, fsubkVal,
      spinMultiplicity, prefactor, summand1, summand2},
653     (
654       {S, L}   = FindSL[NKSL];
655       {Sp, Lp} = FindSL[NKSLp];
656       terms = AllowedNKSLTerms[numE];
657       (* sum for summand1 is over terms with same spin *)
658       spinMultiplicity  = 2*S + 1;
659       termsWithSameSpin = StringCases[terms, ToString[
      spinMultiplicity] ~~ __];
660       termsWithSameSpin = Flatten[termsWithSameSpin];
661       If[Not[{S, L} == {Sp, Lp}],
662         Return[0]
663       ];
664       prefactor = 1/2 * Abs[Ck[orbital, k]]^2;
665       summand1 = Sum[(
666           ReducedUkTable[{numE, orbital, SL, NKSL,  k}] *
667           ReducedUkTable[{numE, orbital, SL, NKSLp, k}]
668           ),
669         {SL, termsWithSameSpin}
670       ];
671       summand1 = 1 / TPO[L] * summand1;
672       summand2 = (
673         KroneckerDelta[NKSL, NKSLp] *
674           (numE *(4*orbital + 2 - numE)) /
675           ((2*orbital + 1) * (4*orbital + 1))
676         );
677       fsubkVal = prefactor*(summand1 - summand2);
678       Return[fsubkVal];
679     )
680   ];
681
682   fsupk::usage = "fsupk[numE, orbital, SL, SLp, k] gives the
        superscripted Slater integral f^k = Subscript[f, k] * Subscript[D,
         k].";
683   fsupk[numE_, orbital_, NKSL_, NKSLp_ ,k_]:= (
```

```
    Dk[k] * fsubk[numE, orbital, NKSL, NKSLp, k]
    )

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.";
Dk[k_] := {1, 225, 1089, 184041/25}[[k/2+1]];

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}];
  )
];

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},
```

```
723     (
724       orbital = 3;
725       Which[
726         OptionValue["Coefficients"] == "Slater",
727         (
728           fsub0 = fsubk[numE, orbital, NKSL, NKSLp, 0];
729           fsub2 = fsubk[numE, orbital, NKSL, NKSLp, 2];
730           fsub4 = fsubk[numE, orbital, NKSL, NKSLp, 4];
731           fsub6 = fsubk[numE, orbital, NKSL, NKSLp, 6];
732           eMatrixVal = fsub0*F0 + fsub2*F2 + fsub4*F4 + fsub6*F6;
733         ),
734         OptionValue["Coefficients"] == "Racah",
735         (
736           fsup0 = fsupk[numE, orbital, NKSL, NKSLp, 0];
737           fsup2 = fsupk[numE, orbital, NKSL, NKSLp, 2];
738           fsup4 = fsupk[numE, orbital, NKSL, NKSLp, 4];
739           fsup6 = fsupk[numE, orbital, NKSL, NKSLp, 6];
740           esub0 = fsup0;
741           esub1 = 9/7*fsup0 +   1/42*fsup2 +   1/77*fsup4 +  1/462*
    fsup6;
742           esub2 =             143/42*fsup2 - 130/77*fsup4 + 35/462*
    fsup6;
743           esub3 =              11/42*fsup2 + 4/77*fsup4   -  7/462*
    fsup6;
744           eMatrixVal = esub0*E0 + esub1*E1 + esub2*E2 + esub3*E3;
745         )
746       ];
747       Return[eMatrixVal];
748     )
749   ];
750
751   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.";
752   Options[GenerateElectrostaticTable] = {"Export" -> True, "
      Coefficients" -> "Slater"};
753   GenerateElectrostaticTable[numEmax_Integer:7, OptionsPattern[]]:= (
754     ElectrostaticTable = Table[
755       {numE, SL, SpLp} -> SimplifyFun[Electrostatic[{numE, SL, SpLp},
      "Coefficients" -> OptionValue["Coefficients"]]],
756       {numE, 1, numEmax},
757       {SL, AllowedNKSLTerms[numE]},
758       {SpLp, AllowedNKSLTerms[numE]}
759     ];
760     ElectrostaticTable = Association[Flatten[ElectrostaticTable]];
761     If[OptionValue["Export"],
762       Export[FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"
      }],
763       ElectrostaticTable];
764     ];
765     Return[ElectrostaticTable];
766   );
767
```

```
768 (* ###################### Electrostatic ###################### *)
769 (* ########################################################## *)
770
771 (* ########################################################## *)
772 (* ########################## Bases ########################## *)
773
774 BasisTableGenerator::usage = "BasisTableGenerator[numE] returns an
      association whose keys are triples of the form {numE, J} and whose
       values are lists having the basis elements that correspond to {
      numE, J}.";
775 BasisTableGenerator[numE_]:=Module[
776   {energyStatesTable, allowedJ, J, Jp},
777   (
778     energyStatesTable = <||>;
779     allowedJ = AllowedJ[numE];
780     Do[
781     (
782       energyStatesTable[{numE, J}] = EnergyStates[numE, J];
783     ),
784     {Jp, allowedJ},
785     {J,  allowedJ}];
786     Return[energyStatesTable]
787   )
788 ];
789
790 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}.
791 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.
792 ";
793 Options[BasisLSJMJ] = {"AsAssociation" -> False};
794 BasisLSJMJ[numE_, OptionsPattern[]]:=Module[
795   {energyStatesTable, basis, idx1},
796   (
797     energyStatesTable = BasisTableGenerator[numE];
798     basis = Table[
799       energyStatesTable[{numE, AllowedJ[numE][[idx1]]}],
800       {idx1, 1, Length[AllowedJ[numE]]}];
801     basis = Flatten[basis, 1];
802     If[OptionValue["AsAssociation"],
803       (
804         Js    = AllowedJ[numE];
805         basis = Table[(J -> Select[basis, #[[2]] == J &]), {J, Js
      }];
806         basis = Association[basis];
807       )
808     ];
809     Return[basis]
810   )
```

```
811 ];
812
813 BasisLSJ::usage="BasisLSJ[numE] returns the intermediate coupling
      basis L-S-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}.
814 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.
815  ";
816 Options[BasisLSJ]={"AsAssociation"->False};
817 BasisLSJ[numE_,OptionsPattern[]]:=Module[
818   {Js,basis},
819   (
820     Js    = AllowedJ[numE];
821     basis = BasisLSJMJ[numE,"AsAssociation"->False];
822     basis = DeleteDuplicates[{#[[1]],#[[2]]}& /@ basis];
823     If[OptionValue["AsAssociation"],
824       (
825         basis = Association @ Table[(J->Select[basis,#[[2]]==J&]),{
      J,Js}]
826       )
827     ];
828     Return[basis];
829   )
830 ];
831
832 (* ########################## Bases ########################### *)
833 (* ############################################################ *)
834
835 (* ############################################################ *)
836 (* ########### Coefficients of Fracional Parentage ########### *)
837
838 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.";
839 Options[GenerateCFP] = {"Export" -> True, "PhaseFunction"-> "NK"};
840 GenerateCFP[OptionsPattern[]]:= (
841   CFP = Table[
842     {numE, NKSL} -> First[CFPTerms[numE, NKSL]],
843     {numE, 1, 7},
844     {NKSL, AllowedNKSLTerms[numE]}];
845   CFP = Association[CFP];
846   (* Go all the way to f14 *)
847   CFP = CFPExpander["Export" -> False, "PhaseFunction"->
      OptionValue["PhaseFunction"]];
848   If[OptionValue["Export"],
849     Export[FileNameJoin[{moduleDir, "data", "CFPs.m"}], CFP];
850   ];
851   Return[CFP];
```

```
852   );
853
854   JuddCFPPhase::usage="Phase between conjugate coefficients of
        fractional parentage according to Velkov's thesis, page 40.";
855   JuddCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
        parentSeniority_, daughterSeniority_]:=Module[
856     {spin, orbital, expo, phase},
857     (
858       {spin, orbital} = {1/2, 3};
859       expo = (
860           (parentS + parentL + daughterS + daughterL) -
861           (orbital + spin) +
862           1/2 * (parentSeniority + daughterSeniority - 1)
863       );
864       phase = Phaser[-expo];
865       Return[phase];
866     )
867   ];
868
869   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).";
870   NKCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
        parentSeniority_, daughterSeniority_]:=Module[
871     {spin, orbital, expo, phase},
872     (
873       {spin, orbital} = {1/2, 3};
874       expo = (
875           (parentS + parentL + daughterS + daughterL) -
876           (orbital + spin)
877       );
878       phase = Phaser[-expo];
879       If[parent == 2*orbital,
880           phase = phase * Phaser[(daughterSeniority-1)/2]];
881       Return[phase];
882     )
883   ];
884
885   Options[CFPExpander] = {"Export" -> True, "PhaseFunction" -> "NK"};
886   CFPExpander::usage="Using the coefficients of fractional parentage
        up to f7 this function calculates them up to f14.
887   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.";
888   CFPExpander[OptionsPattern[]]:=Module[
889     {orbital, halfFilled, fullShell, parentMax, PhaseFun,
890     complementaryCFPs, daughter, conjugateDaughter,
891     conjugateParent, parentTerms, daughterTerms,
892     parentCFPs, daughterSeniority, daughterS, daughterL,
893     parentCFP, parentTerm, parentCFPval,
894     parentS, parentL, parentSeniority, phase, prefactor,
```

```
895     newCFPval, key, extendedCFPs, exportFname},
896     (
897       orbital    = 3;
898       halfFilled = 2 * orbital + 1;
899       fullShell  = 2 * halfFilled;
900       parentMax  = 2 * orbital;
901
902       PhaseFun  = <|
903           "Judd" -> JuddCFPPhase,
904           "NK" -> NKCFPPhase|>[OptionValue["PhaseFunction"]];
905       PrintTemporary["Calculating CFPs using the phase system from ",
      PhaseFun];
906       (* Initialize everything with lists to be filled in the next Do
     *)
907       complementaryCFPs =
908           Table[
909           ({numE, term} -> {term}),
910           {numE, halfFilled + 1, fullShell - 1, 1},
911           {term, AllowedNKSLTerms[numE]
912           }];
913       complementaryCFPs = Association[Flatten[complementaryCFPs]];
914       Do[(
915           daughter          = parent + 1;
916           conjugateDaughter = fullShell - parent;
917           conjugateParent   = conjugateDaughter - 1;
918           parentTerms       = AllowedNKSLTerms[parent];
919           daughterTerms     = AllowedNKSLTerms[daughter];
920           Do[
921           (
922               parentCFPs             = Rest[CFP[{daughter,
     daughterTerm}]];
923               daughterSeniority      = Seniority[daughterTerm];
924               {daughterS, daughterL} = FindSL[daughterTerm];
925               Do[
926               (
927                   {parentTerm, parentCFPval} = parentCFP;
928                   {parentS, parentL}         = FindSL[parentTerm];
929                   parentSeniority            = Seniority[parentTerm];
930                   phase = PhaseFun[parent, parentS, parentL,
931                                    daughterS, daughterL,
932                                    parentSeniority, daughterSeniority
     ];
933                   prefactor = (daughter * TPO[daughterS, daughterL])
     /
934                               (conjugateDaughter * TPO[parentS,
     parentL]);
935                   prefactor = Sqrt[prefactor];
936                   newCFPval = phase * prefactor * parentCFPval;
937                   key = {conjugateDaughter, parentTerm};
938                   complementaryCFPs[key] = Append[complementaryCFPs[
     key], {daughterTerm, newCFPval}]
939               ),
940               {parentCFP, parentCFPs}
941               ]
942           ),
```

```
943             {daughterTerm, daughterTerms}
944             ]
945             ),
946         {parent, 1, parentMax}
947         ];
948 
949         complementaryCFPs[{14, "1S"}] = {"1S", {"2F",1}};
950         extendedCFPs          = Join[CFP, complementaryCFPs];
951         If[OptionValue["Export"];,
952         (
953             exportFname = FileNameJoin[{moduleDir, "data", "
    CFPs_extended.m"}];
954             Print["Exporting to ", exportFname];
955             Export[exportFname, extendedCFPs];
956         )
957         ];
958         Return[extendedCFPs];
959     )
960     ];
961 
962     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.
963     The data being parsed here is the file attachment B1F_ALL.TXT which
       comes from Velkov's thesis.";
964     Options[GenerateCFPTable] = {"Export" -> True};
965     GenerateCFPTable[OptionsPattern[]]:=Module[
966       {rawText, rawLines, leadChar, configIndex, line, daughter,
967       lineParts, numberCode, parsedNumber, toAppend, CFPTablefname},
968       (
969         CleanWhitespace[string_]     := StringReplace[string,
      RegularExpression["\\s+"]->" "];
970         AddSpaceBeforeMinus[string_] := StringReplace[string,
      RegularExpression["(?<!\\s)-"]->" -"];
971         ToIntegerOrString[list_]     := Map[If[StringMatchQ[#,
      NumberString], ToExpression[#], #] &, list];
972         CFPTable      = ConstantArray[{},7];
973         CFPTable[[1]] = {{"2F",{"1S",1}}};
974 
975         (* Cleaning before processing is useful *)
976         rawText  = Import[FileNameJoin[{moduleDir, "data", "B1F_ALL.TXT
      "}]];
977         rawLines = StringTrim/@StringSplit[rawText,"\n"];
978         rawLines = Select[rawLines,#!=""&];
979         rawLines = CleanWhitespace/@rawLines;
980         rawLines = AddSpaceBeforeMinus/@rawLines;
981 
982         Do[(
983           (* the first character can be used to identify the start of a
       block *)
984           leadChar=StringTake[line,{1}];
985           (* ..FN, N is at position 50 in that line *)
986           If[leadChar=="[",
987           (
```

```
988             configIndex=ToExpression[StringTake[line,{50}]];
989             Continue[];
990           )
991           ];
992           (* Identify which daughter term is being listed *)
993           If[StringContainsQ[line,"[DAUGHTER TERM]"],
994             daughter=StringSplit[line,"["][[1]];
995             CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{
    daughter}];
996             Continue[];
997           ];
998           (* Once we get here we are already parsing a row with
    coefficient data *)
999           lineParts    = StringSplit[line," "];
1000          parent       = lineParts[[1]];
1001          numberCode   = ToIntegerOrString[lineParts[[3;;]]];
1002          parsedNumber = SquarePrimeToNormal[numberCode];
1003          toAppend     = {parent,parsedNumber};
1004          CFPTable[[configIndex]][[-1]] = Append[CFPTable[[configIndex
    ]][[-1]], toAppend]
1005        ),
1006        {line,rawLines}];
1007        If[OptionValue["Export"],
1008          (
1009          CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"
    }];
1010          Export[CFPTablefname, CFPTable];
1011          )
1012        ];
1013        Return[CFPTable];
1014     )
1015   ];
1016
1017   GenerateCFPAssoc::usage = "GenerateCFPAssoc[export] converts the
      coefficients of fractional parentage into an association in which
      zero values are explicit. If \"Export\" is set to True, the
      association is exported to the file /data/CFPAssoc.m. This
      function requires that the association CFP be defined.";
1018  Options[GenerateCFPAssoc] = {"Export" -> True};
1019  GenerateCFPAssoc[OptionsPattern[]]:= (
1020    CFPAssoc = Association[];
1021    Do[
1022      (daughterTerms = AllowedNKSLTerms[numE];
1023      parentTerms    = AllowedNKSLTerms[numE - 1];
1024      Do[
1025        (
1026        cfps = CFP[{numE, daughter}];
1027        cfps = cfps[[2 ;;]];
1028        parents = First /@ cfps;
1029        Do[
1030          (
1031          key = {numE, daughter, parent};
1032          cfp = If[
1033            MemberQ[parents, parent],
1034            (
```

```
1035                idx = Position[parents, parent][[1, 1]];
1036                cfps[[idx]][[2]]
1037              ),
1038              0
1039              ];
1040            CFPAssoc[key] = cfp;
1041            ),
1042            {parent, parentTerms}
1043            ]
1044          ),
1045          {daughter, daughterTerms}
1046          ]
1047        ),
1048        {numE, 1, 14}
1049        ];
1050      If[OptionValue["Export"],
1051        (
1052        CFPAssocfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"
      }];
1053        Export[CFPAssocfname, CFPAssoc];
1054        )
1055      ];
1056      Return[CFPAssoc];
1057    );
1058
1059    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.
1060    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.
1061    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.
1062    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}, ...}.
1063    Only the one-body coefficients for f-electrons are provided.
1064    In all cases it must be that 1 <= n <= 7.
1065    ";
1066    CFPTerms[numE_] := Part[CFPTable, numE]
1067    CFPTerms[numE_, SL_]:=Module[
1068      {NKterms, CFPconfig},
1069      (
1070        NKterms   = {{}};
1071        CFPconfig = CFPTable[[numE]];
1072        Map[
1073          If[StringFreeQ[First[#], SL],
1074            Null,
1075            NKterms = Join[NKterms, {#}, 1]
1076          ] &,
1077        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];
    )
  ];

  GenerateSpinOrbitTable::usage = "GenerateSpinOrbitTable[nmax]
    computes the matrix values 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 subfolder for
     the folder in which this package is in. 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"],
      (
        Print["Exporting to file "<>ToString[exportFname]];
        Export[exportFname, SpinOrbitTable];
      )
    ];
    Return[SpinOrbitTable];
  )
];

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

(* ############################################################## *)
(* ################## Three Body Operators ###################### *)

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 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 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];
  );

  GenerateThreeBodyTables::usage="This function generates the matrix elements for the three body operators using the coefficients of fractional parentage, including those beyond f^7.";
  Options[GenerateThreeBodyTables] = {"Export" -> False};
  GenerateThreeBodyTables[nmax_Integer : 14, OptionsPattern[]] := (
    tiKeys = {"t_{2}", "t_{2}^{'}", "t_{3}", "t_{4}", "t_{6}", "t_{7}",
      "t_{8}", "t_{11}", "t_{11}^{'}", "t_{12}", "t_{14}", "t_{15}",
      "t_{16}", "t_{17}", "t_{18}", "t_{19}"};
    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];
```

```
1268        Return[val];
1269      );
1270      (* ti: This is the implementation of formula (2) in Judd & Suskin
           1984. It computes the matrix elements of ti in f^n by using the
          matrix elements in f3 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. *)
1271      Options[ti] = {"Fast" -> True};
1272      ti[nE_, SL_, SpLp_, tiKey_, opOrder_ : 3, OptionsPattern[]]:=
          Module[
1273        {
1274          nn, S, L, Sp, Lp,
1275          cfpSL, cfpSpLp,
1276          parentSL, parentSpLp, tnk, tnks
1277        },
1278        (
1279          {S, L}   = FindSL[SL];
1280          {Sp, Lp} = FindSL[SpLp];
1281          fast     = OptionValue["Fast"];
1282          numH = 14 - nE;
1283          If[fast && Not[MemberQ[{"t_{2}","t_{11}"},tiKey]] && nE > 7,
1284            Return[-tktable[{numH, SL, SpLp, tiKey}]]
1285          ];
1286          If[(S == Sp && L == Lp),
1287          (
1288            cfpSL   = CFP[{nE, SL}];
1289            cfpSpLp = CFP[{nE, SpLp}];
1290            tnks = Table[(
1291                parentSL   = cfpSL[[nn, 1]];
1292                parentSpLp = cfpSpLp[[mm, 1]];
1293                cfpSL[[nn, 2]] * cfpSpLp[[mm, 2]] *
1294                tktable[{nE - 1, parentSL, parentSpLp, tiKey}]
1295                ),
1296                {nn, 2, Length[cfpSL]},
1297                {mm, 2, Length[cfpSpLp]}
1298                ];
1299            tnk = Total[Flatten[tnks]];
1300            ),
1301          tnk = 0;
1302          ];
1303          Return[ nE / (nE - opOrder) * tnk];
1304        )
1305      ];
1306      (*Calculate the matrix elements of t^i for n up to nmax*)
1307      tktable = <||>;
1308      Do[(
1309        Do[(
1310          tkValue = Which[numE <= 2,
1311            (*Initialize n=1,2 with zeros*)
1312            0,
1313            numE == 3,
1314            (*Grab matrix elem in f^3 from Judd 1984*)
1315            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, nmax}
];

(* Now use those 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]];
```

```
1366            reps = ConstantArray[0, Length[TSymbols]];
1367            reps[[notNullPosition]] = 1;
1368            rep = AssociationThread[TSymbols -> reps];
1369            notNullSymbol -> Association[(singleThreeBodyTable /. rep)]
1370            ),
1371          {notNullSymbol, TSymbols}
1372          ];
1373        singleThreeBodyTables = Association[singleThreeBodyTables];
1374        numE -> singleThreeBodyTables),
1375        {numE, 1, 7}
1376    ];
1377
1378    ThreeBodyTables = Association[ThreeBodyTables];
1379    If[OptionValue["Export"],
1380      (
1381        threeBodyTablefname = FileNameJoin[{moduleDir, "data", "
       ThreeBodyTable.m"}];
1382        Export[threeBodyTablefname, ThreeBodyTable];
1383        threeBodyTablesfname = FileNameJoin[{moduleDir, "data", "
       ThreeBodyTables.m"}];
1384        Export[threeBodyTablesfname, ThreeBodyTables];
1385      )
1386     ];
1387    Return[{ThreeBodyTable, ThreeBodyTables}];
1388    );
1389
1390    ScalarOperatorProduct::usage="ScalarOperatorProduct[op1, op2, numE]
       calculated the innerproduct between the two scalar operators op1
       and op2.";
1391    ScalarOperatorProduct[op1_, op2_, numE_]:=Module[
1392      {terms, S, L, factor, term1, term2},
1393      (
1394        terms = AllowedNKSLTerms[numE];
1395        Simplify[
1396          Sum[(
1397            {S, L} = FindSL[term1];
1398            factor = TPO[S, L];
1399            factor * op1[{term1, term2}] * op2[{term2, term1}]
1400            ),
1401          {term1, terms},
1402          {term2, terms}
1403          ]
1404        ]
1405      )
1406    ];
1407
1408    (* ################## Three Body Operators ##################### *)
1409    (* ############################################################ *)
1410
1411    (* ############################################################ *)
1412    (* ################### Reduced SOO and ECSO ################### *)
1413
1414    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.
```

```
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.
";
ReducedT11inf2[SL_, SpLp_] := Module[
  {T11inf2},
  (
    T11inf2 = <|
      {"1S", "3P"} -> 6 M0 + 2 M2 + 10/11 M4,
      {"3P", "3P"} -> -36 M0 - 72 M2 - 900/11 M4,
      {"3P", "1D"} -> -Sqrt[(2/15)] (27 M0 + 14 M2 + 115/11 M4),
      {"1D", "3F"} -> Sqrt[2/5] (23 M0 + 6 M2 - 195/11 M4),
      {"3F", "3F"} -> 2 Sqrt[14] (-15 M0 - M2 + 10/11 M4),
      {"3F", "1G"} -> Sqrt[11] (-6 M0 + 64/33 M2 - 1240/363 M4),
      {"1G", "3H"} -> Sqrt[2/5] (39 M0 - 728/33 M2 - 3175/363 M4),
      {"3H", "3H"} -> 8/Sqrt[55] (-132 M0 + 23 M2 + 130/11 M4),
      {"3H", "1I"} -> Sqrt[26] (-5 M0 - 30/11 M2 - 375/1573 M4)
      |>;
    Which[
      MemberQ[Keys[T11inf2],{SL,SpLp}],
        Return[T11inf2[{SL,SpLp}]],
      MemberQ[Keys[T11inf2],{SpLp,SL}],
        Return[T11inf2[{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]
      ]
    )
];

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 1966 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.
Rigurously 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] +
```

```
1501            Reducedt11inf2[SL, SpLp] -
1502            a13 / 6 * matElement
1503        );
1504        redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
1505        Return[redSOOandECSOinf2];
1506      )
1507    ];
1508
1509    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.
1510  ";
1511  ReducedSOOandECSOinfn[numE_, SL_, SpLp_]:=Module[
1512    {spin, orbital, t, S, L, Sp, Lp, idx1, idx2, cfpSL, cfpSpLp,
      parentSL, Sb, Lb, Sbp, Lbp, parentSpLp, funval},
1513    (
1514      {spin, orbital} = {1/2, 3};
1515      {S, L}   = FindSL[SL];
1516      {Sp, Lp} = FindSL[SpLp];
1517      t = 1;
1518      cfpSL    = CFP[{numE, SL}];
1519      cfpSpLp  = CFP[{numE, SpLp}];
1520      funval   = Sum[
1521        (
1522          parentSL = cfpSL[[idx2, 1]];
1523          parentSpLp = cfpSpLp[[idx1, 1]];
1524          {Sb, Lb}   = FindSL[parentSL];
1525          {Sbp, Lbp} = FindSL[parentSpLp];
1526          phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
1527          (
1528            phase *
1529            cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
1530            SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
1531            SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
1532            SOOandECSOLSTable[{numE - 1, parentSL, parentSpLp}]
1533          )
1534        ),
1535      {idx1, 2, Length[cfpSpLp]},
1536      {idx2, 2, Length[cfpSL]}
1537      ];
1538      funval *= numE / (numE - 2) * Sqrt[TPO[S, Sp, L, Lp]];
1539      Return[funval];
1540    )
1541  ];
1542
1543  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.";
1544 Options[GenerateSOOandECSOLSTable] = {"Progress" -> True, "Export"
    -> True};
1545 GenerateSOOandECSOLSTable[nmax_Integer, OptionsPattern[]]:= (
1546   If[And[OptionValue["Progress"], frontEndAvailable],
1547     (
1548       numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
    numE]]^2, {numE, 1, nmax}]];
1549       counters   = Association[Table[numE->0, {numE, 1, nmax}]];
1550       totalIters = Total[Values[numItersai[[1;;nmax]]]];
1551       template1  = StringTemplate["Iteration `numiter` of `
    totaliter`"];
1552       template2  = StringTemplate["`remtime` min remaining"];
    template3 = StringTemplate["Iteration speed = `speed` ms/it"];
1553       template4  = StringTemplate["Time elapsed = `runtime` min"];
1554       progBar = PrintTemporary[
1555         Dynamic[
1556           Pane[
1557             Grid[{
1558                   {Superscript["f", numE]},
1559                   {template1[<|"numiter"->numiter, "totaliter"->
    totalIters|>]},
1560                   {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
1561                   {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
    ]], 0.1]|>]},
1562                   {template3[<|"speed"->Round[QuantityMagnitude[Now
    -startTime, "ms"]/(numiter), 0.01]|>]}, {ProgressIndicator[Dynamic
    [numiter], {1, totalIters}]}
1563                 },
1564                 Frame->All
1565             ],
1566             Full,
1567             Alignment->Center
1568           ]
1569         ]
1570       ];
1571     )
1572   ];
1573   SOOandECSOLSTable = <||>;
1574   numiter   = 1;
1575   startTime = Now;
1576   Do[
1577     (
1578       numiter+= 1;
1579       SOOandECSOLSTable[{numE, SL, SpLp}] = Which[
1580         numE==1,
1581         0,
1582         numE==2,
1583         SimplifyFun[ReducedSOOandECSOinf2[SL, SpLp]],
```

```
1584            True,
1585            SimplifyFun[ReducedSOOandECSOinfn[numE,  SL, SpLp]]
1586          ];
1587        ),
1588      {numE, 1, nmax},
1589      {SL, AllowedNKSLTerms[numE]},
1590      {SpLp, AllowedNKSLTerms[numE]}
1591      ];
1592      If[And[OptionValue["Progress"], frontEndAvailable],
1593        NotebookDelete[progBar]];
1594      If[OptionValue["Export"],
1595        (fname = FileNameJoin[{moduleDir, "data", "
      ReducedSOOandECSOLSTable.m"}];
1596        Export[fname, SOOandECSOLSTable];
1597        )
1598      ];
1599      Return[SOOandECSOLSTable];
1600    );
1601
1602    (* ################## Reduced SOO and ECSO ################## *)
1603    (* ############################################################ *)
1604
1605    (* ############################################################ *)
1606    (* ###################### Spin-Spin ############################ *)
1607
1608    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.
1609    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.
1610    ";
1611    ReducedT22inf2[SL_, SpLp_]:=Module[
1612      {statePosition, PsiPsipStates, m0, m2, m4, Tkk2m},
1613      (
1614        T22inf2 = <|
1615        {"3P", "3P"} -> -12 M0 - 24 M2 - 300/11 M4,
1616        {"3P", "3F"} -> 8/Sqrt[3] (3 M0 + M2 - 100/11 M4),
1617        {"3F", "3F"} -> 4/3 Sqrt[14] (-M0 + 8 M2 - 200/11 M4),
1618        {"3F", "3H"} -> 8/3 Sqrt[11/2] (2 M0 - 23/11 M2 - 325/121 M4),
1619        {"3H", "3H"} -> 4/3 Sqrt[143] (M0 - 34/11 M2 - (1325/1573) M4)
1620        |>;
1621        Which[
1622          MemberQ[Keys[T22inf2],{SL,SpLp}],
1623            Return[T22inf2[{SL,SpLp}]],
1624          MemberQ[Keys[T22inf2],{SpLp,SL}],
1625            Return[T22inf2[{SpLp,SL}]],
1626          True,
1627            Return[0]
1628        ]
1629      )
1630    ];
1631
1632    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 the
    operator corresponding to the inter-electron between spin.
1633  It does this 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.\"
1634  ";
1635  ReducedT22infn[numE_, SL_, SpLp_]:=Module[
1636    {spin, orbital, t, idx1, idx2, S, L, Sp, Lp, cfpSL, cfpSpLp,
    parentSL, parentSpLp, Sb, Lb, Tnkk, phase, Sbp, Lbp},
1637    (
1638      {spin, orbital} = {1/2, 3};
1639      {S, L}   = FindSL[SL];
1640      {Sp, Lp} = FindSL[SpLp];
1641      t = 2;
1642      cfpSL    = CFP[{numE, SL}];
1643      cfpSpLp  = CFP[{numE, SpLp}];
1644      Tnkk = Sum[(
1645        parentSL   = cfpSL[[idx2, 1]];
1646        parentSpLp = cfpSpLp[[idx1, 1]];
1647        {Sb, Lb}   = FindSL[parentSL];
1648        {Sbp, Lbp} = FindSL[parentSpLp];
1649        phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
1650        (
1651          phase *
1652          cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
1653          SixJay[{S, t, Sp}, {Sbp, spin, Sb}] *
1654          SixJay[{L, t, Lp}, {Lbp, orbital, Lb}] *
1655          T22Table[{numE - 1, parentSL, parentSpLp}]
1656        )
1657      ),
1658      {idx1, 2, Length[cfpSpLp]},
1659      {idx2, 2, Length[cfpSL]}
1660      ];
1661      Tnkk *= numE / (numE - 2) * Sqrt[TPO[S,Sp,L,Lp]];
1662      Return[Tnkk];
1663    )
1664  ];
1665
1666  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.
1667  This is an intermediate step to the calculation of the reduced
    matrix elements of the spin-spin operator.";
1668  Options[GenerateT22Table] = {"Export" -> True, "Progress" -> True};
1669  GenerateT22Table[nmax_Integer, OptionsPattern[]]:= (
1670    If[And[OptionValue["Progress"], frontEndAvailable],
1671      (
1672        numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
```

```
    numE]]^2, {numE, 1, nmax}]];
1673        counters = Association[Table[numE->0, {numE, 1, nmax}]];
1674        totalIters = Total[Values[numItersai[[1;;nmax]]]];
1675        template1 = StringTemplate["Iteration `numiter` of `totaliter
    `"];
1676        template2 = StringTemplate["`remtime` min remaining"];
    template3 = StringTemplate["Iteration speed = `speed` ms/it"];
1677        template4 = StringTemplate["Time elapsed = `runtime` min"];
1678        progBar = PrintTemporary[
1679          Dynamic[
1680            Pane[
1681              Grid[{{Superscript["f", numE]},
1682                    {template1[<|"numiter"->numiter, "totaliter"->
    totalIters|>]},
1683                    {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
1684                    {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
    ]], 0.1]|>]},
1685                    {template3[<|"speed"->Round[QuantityMagnitude[Now
    -startTime, "ms"]/(numiter), 0.01]|>]},
1686                    {ProgressIndicator[Dynamic[numiter], {1,
    totalIters}]}},
1687                    Frame->All],
1688                    Full,
1689                    Alignment->Center]
1690                ]
1691              ];
1692      )
1693    ];
1694    T22Table = <||>;
1695    startTime = Now;
1696    numiter = 1;
1697    Do[
1698      (
1699        numiter+= 1;
1700        T22Table[{numE, SL, SpLp}] = Which[
1701          numE==1,
1702          0,
1703          numE==2,
1704          SimplifyFun[ReducedT22inf2[SL, SpLp]],
1705          True,
1706          SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
1707        ];
1708      ),
1709    {numE, 1, nmax},
1710    {SL,   AllowedNKSLTerms[numE]},
1711    {SpLp, AllowedNKSLTerms[numE]}
1712    ];
1713    If[And[OptionValue["Progress"],frontEndAvailable],
1714      NotebookDelete[progBar]
1715    ];
1716    If[OptionValue["Export"],
1717      (
1718        fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"
```

```
    }];
1719          Export[fname, T22Table];
1720        )
1721     ];
1722     Return[T22Table];
1723   );
1724
1725   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.
1726   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.\"
1727   \".
1728   ";
1729   SpinSpin[numE_, SL_, SpLp_, J_]:=Module[
1730     {S, L, Sp, Lp, α, val},
1731     (
1732       α = 2;
1733       {S, L} = FindSL[SL];
1734       {Sp, Lp} = FindSL[SpLp];
1735       val = (
1736               Phaser[Sp + L + J] *
1737               SixJay[{Sp, Lp, J}, {L, S, α}] *
1738               T22Table[{numE, SL, SpLp}]
1739             );
1740       Return[val]
1741     )
1742   ];
1743
1744   GenerateSpinSpinTable::usage="GenerateSpinSpinTable[nmax] generates
        the 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 {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.";
1745   Options[GenerateSpinSpinTable] = {"Export"->False};
1746   GenerateSpinSpinTable[nmax_, OptionsPattern[]] :=
1747     (
1748       SpinSpinTable = <||>;
1749       PrintTemporary[Dynamic[numE]];
1750       Do[
1751         SpinSpinTable[{numE, SL, SpLp, J}] = (SpinSpin[numE, SL, SpLp
        , J]);,
1752       {numE, 1, nmax},
1753       {J, MinJ[numE], MaxJ[numE]},
1754       {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
1755       {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
1756       ];
```

```
      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 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.";
1792  Options[GenerateSOOandECSOTable] = {"Export"->False}
1793  GenerateSOOandECSOTable[nmax_, OptionsPattern[]]:= (
1794    SOOandECSOTable = <||>;
1795    Do[
1796      SOOandECSOTable[{numE, SL, SpLp, J}] = (SOOandECSO[numE, SL,
      SpLp, J] /. Prescaling);,
1797    {numE, 1, nmax},
1798    {J, MinJ[numE], MaxJ[numE]},
1799    {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
1800    {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
1801    ];
1802    If[OptionValue["Export"],
1803    (
1804      fname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
1805      Export[fname, SOOandECSOTable];
1806    )
1807    ];
1808    Return[SOOandECSOTable];
1809  );
1810
1811  (* ## Spin-Other-Orbit and Electrostatically-Correlated-Spin-Orbit
      ## *)
1812  (* ################################################################ *)
1813
1814  (* ################################################################ *)
1815  (* ################### Magnetic Interactions ################### *)
1816
1817  MagneticInteractions::usage="MagneticInteractions[{numE, SLJ, SLJp,
      J}] returns the matrix element of the magnetic interaction
    between the terms SLJ and SLJp in the f^numE configuration. The
    interaction is given by the sum of the spin-spin, the spin-other-
    orbit,  and the electrostatically-correlated-spin-orbit
    interactions.
1818  The part corresponding to the spin-spin interaction is provided by
    SpinSpin[{numE, SLJ, SLJp, J}].
1819  The part corresponding to SOO and ECSO is provided by the function
    SOOandECSO[{numE, SLJ, SLJp, J}].
1820  The function requires chenDeltas to be loaded into the session.
1821  The option \"ChenDeltas\" can be used to include or exclude the
    Chen deltas from the calculation. The default is to exclude them."
    ;
1822  Options[MagneticInteractions] = {"ChenDeltas" -> False};
1823  MagneticInteractions[{numE_, SLJ_, SLJp_, J_}, OptionsPattern[]] :=
1824    (
1825      key = {numE, SLJ, SLJp, J};
1826      ss = \[Sigma]SS * SpinSpinTable[key];
1827      sooandecso = SOOandECSOTable[key];
1828      total = ss + sooandecso;
1829      total = SimplifyFun[total];
1830      If[
1831        Not[OptionValue["ChenDeltas"]],
1832        Return[total]
1833      ];
1834      (* In the type A errors the wrong values are different *)
```

```
      If[MemberQ[Keys[chenDeltas["A"]], {numE, SLJ, SLJp}],
        (
          {S, L} = FindSL[SLJ];
          {Sp, Lp} = FindSL[SLJp];
          phase   = Phaser[Sp + L + J];
          Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
          Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
          {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SLJ
    , SLJp}]["wrong"];
          total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
          total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
          total  = total /. Prescaling;
          total  = wChErrA * total + (1 - wChErrA) * (ss + sooandecso
    )
        )
      ];
      (* In the type B errors the wrong values are zeros all around
    *)
      If[MemberQ[chenDeltas["B"], {numE, SLJ, SLJp}],
        (
          {S, L} = FindSL[SLJ];
          {Sp, Lp} = FindSL[SLJp];
          phase = Phaser[Sp + L + J];
          Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
          Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
          {M0v, M2v, M4v, P2v, P4v, P6v} = {0, 0, 0, 0, 0, 0};
          total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
          total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
          total  = total /. Prescaling;
          total  = wChErrB * total + (1 - wChErrB) * (ss + sooandecso
    )
        )
      ];
      Return[total];
    )

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

  (* ########################################################### *)
  (* ###################### 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[TP0[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]
    gives the general expression for the matrix element of the crystal
     field Hamiltonian parametrized with Bqk and Sqk coefficients as a
     sum over spherical harmonics Cqk.
  Sometimes this expression only includes Bqk coefficients, see for
    example eqn 6-2 in Wybourne (1965), but one may also split the
    coefficient into real and imaginary parts as is done here, in an
    expression that is patently Hermitian.";
  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}
```

```
1933      ]
1934    )
1935
1936    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.";
1937    TotalCFIters[i_, j_] := (
1938      numIters = {196, 8281, 132496, 1002001, 4008004, 9018009,
          11778624};
1939      Return[Total[numIters[[i ;; j]]]];
1940      )
1941
1942    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.";
1943    Options[GenerateCrystalFieldTable] = {"Export" -> False, "Progress"
          -> True, "Compress" -> True}
1944    GenerateCrystalFieldTable[numEs_List:{1,2,3,4,5,6,7},
          OptionsPattern[]]:= (
1945      ExportFun =
1946      If[OptionValue["Compress"],
1947        ExportMZip,
1948        Export
1949      ];
1950      numiter = 1;
1951      template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
1952      template2 = StringTemplate["`remtime` min remaining"];
1953      template3 = StringTemplate["Iteration speed = `speed` ms/it"];
1954      template4 = StringTemplate["Time elapsed = `runtime` min"];
1955      totalIter = Total[TotalCFIters[#, #] & /@ numEs];
1956      freebies = 0;
1957      startTime = Now;
1958      If[And[OptionValue["Progress"], frontEndAvailable],
1959        progBar = PrintTemporary[
1960          Dynamic[
1961            Pane[
1962              Grid[
1963                {
1964                  {Superscript["f", numE]},
1965                  {template1[<|"numiter" -> numiter, "totaliter" ->
          totalIter|>]},
1966                  {template4[<|"runtime" -> Round[QuantityMagnitude[
          UnitConvert[(Now - startTime), "min"]], 0.1]|>]},
1967                  {template2[<|"remtime" -> Round[QuantityMagnitude[
          UnitConvert[(Now - startTime)/(numiter - freebies) * (totalIter -
          numiter), "min"]], 0.1]|>]},
1968                  {template3[<|"speed" -> Round[QuantityMagnitude[Now -
```

```
     startTime, "ms"]/(numiter-freebies), 0.01]|>]},
1969                 {ProgressIndicator[Dynamic[numiter], {1, totalIter}]}
1970               },
1971             Frame -> All
1972             ],
1973           Full,
1974           Alignment -> Center
1975           ]
1976         ]
1977       ];
1978     ];
1979     Do[
1980       (
1981         exportFname = FileNameJoin[{moduleDir, "data", "
    CrystalFieldTable_f"<>ToString[numE]<>".m"}];
1982         If[FileExistsQ[exportFname],
1983           Print["File exists, skipping ..."];
1984           numiter+=  TotalCFIters[numE, numE];
1985           freebies+= TotalCFIters[numE, numE];
1986           Continue[];
1987         ];
1988         CrystalFieldTable = <||>;
1989         Do[
1990           (
1991             numiter+= 1;
1992             CrystalFieldTable[{numE, NKSL, J, M, NKSLp, Jp, Mp}] =
    CrystalField[numE, NKSL, J, M, NKSLp, Jp, Mp];
1993           ),
1994         {J, MinJ[numE], MaxJ[numE]},
1995         {Jp, MinJ[numE], MaxJ[numE]},
1996         {M, AllowedMforJ[J]},
1997         {Mp, AllowedMforJ[Jp]},
1998         {NKSL , First /@ AllowedNKSLforJTerms[numE, J]},
1999         {NKSLp, First /@ AllowedNKSLforJTerms[numE, Jp]}
2000         ];
2001         If[And[OptionValue["Progress"],frontEndAvailable],
2002           NotebookDelete[progBar]
2003         ];
2004         If[OptionValue["Export"],
2005           (
2006             Print["Exporting to file "<>ToString[exportFname]];
2007             ExportFun[exportFname, CrystalFieldTable];
2008           )
2009         ];
2010       ),
2011     {numE, numEs}
2012     ]
2013   )
2014
2015   (* ##################### Crystal Field ###################### *)
2016   (* ############################################################ *)
2017
2018   (* ############################################################ *)
2019   (* #### Configuration-Interaction via Casimir Operators ###### *)
2020
```

```
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[{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[{SL_, SpLp_}]:=Module[
    {S, L, val},
    (
      {S, L} = FindSL[SL];
      val = (
        If[SL == SpLp,
          CasimirSO3[{SL, SL}] +
          CasimirSO7[{SL, SL}] +
          CasimirG2[{SL, SL}],
          0
        ]
        );
      ElectrostaticConfigInteraction[{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,
```

```
2109      subKron, matValue, eMatrix},
2110      (
2111        NKSLJMs  = AllowedNKSLJMforJTerms[numE, J];
2112        NKSLJMps = AllowedNKSLJMforJTerms[numE, Jp];
2113        eMatrix =
2114          Table[
2115            (*Condition for a scalar matrix op*)
2116            SLterm   =  NKSLJM[[1]];
2117            SpLpterm = NKSLJMp[[1]];
2118            MJ       =  NKSLJM[[3]];
2119            MJp      = NKSLJMp[[3]];
2120            subKron  =
2121              (
2122                KroneckerDelta[J, Jp] *
2123                KroneckerDelta[MJ, MJp]
2124              );
2125            matValue =
2126              If[subKron==0,
2127                0,
2128                  (
2129                    ElectrostaticTable[{numE, SLterm, SpLpterm}] +
2130                    ElectrostaticConfigInteraction[{SLterm, SpLpterm}]
    +
2131                    SpinOrbitTable[{numE, SLterm, SpLpterm, J}] +
2132                    MagneticInteractions[{numE, SLterm, SpLpterm, J}, "
    ChenDeltas" -> OptionValue["ChenDeltas"]] +
2133                    ThreeBodyTable[{numE, SLterm, SpLpterm}]
2134                  )
2135              ];
2136            matValue += CFTable[{numE, SLterm, J, MJ, SpLpterm, Jp, MJp
    }];
2137            matValue,
2138          {NKSLJMp, NKSLJMps},
2139          {NKSLJM , NKSLJMs}
2140          ];
2141      If[OptionValue["Sparse"],
2142        eMatrix = SparseArray[eMatrix]
2143      ];
2144      Return[eMatrix]
2145    )
2146    ];
2147
2148    EnergyStates::usage = "Alias for AllowedNKSLJMforJTerms. At some
        point may be used to redefine states used in basis.";
2149    EnergyStates[numE_, J_]:= AllowedNKSLJMforJTerms[numE, J];
2150
2151    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.";
2152    Options[JJBlockMatrixFileName] = {"FilenameAppendix" -> ""}
2153    JJBlockMatrixFileName[numE_Integer, OptionsPattern[]] := (
2154      fileApp = OptionValue["FilenameAppendix"];
2155      fname = FileNameJoin[{moduleDir,
2156          "hams",
```

```
2157        StringJoin[{"f", ToString[numE], "_JJBlockMatrixTable",
      fileApp ,".m"}]}];
2158     Return[fname];
2159   );
2160
2161   TabulateJJBlockMatrixTable::usage = "TabulateJJBlockMatrixTable[
      numE, I] returns a list with three elements {JJBlockMatrixTable,
      EnergyStatesTable, AllowedM}. JJBlockMatrixTable is an association
       with keys equal to lists of the form {numE, J, Jp}.
      EnergyStatesTable is an association with keys equal to lists of
      the form {numE, J}. AllowedM is another association with keys
      equal to lists of the form {numE, J} and values equal to lists
      equal to the corresponding values of MJ. It's unnecessary (and it
      won't work in this implementation) to give numE > 7 given the
      equivalency between electron and hole configurations.";
2162   Options[TabulateJJBlockMatrixTable] = {"Sparse"->True, "ChenDeltas"
      ->False};
2163   TabulateJJBlockMatrixTable[numE_, CFTable_, OptionsPattern[]]:= (
2164     JJBlockMatrixTable = <||>;
2165     totalIterations = Length[AllowedJ[numE]]^2;
2166     template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
2167     template2 = StringTemplate["`remtime` min remaining"];
2168     template4 = StringTemplate["Time elapsed = `runtime` min"];
2169     numiter   = 0;
2170     startTime = Now;
2171     If[$FrontEnd =!= Null,
2172       (
2173         temp = PrintTemporary[
2174           Dynamic[
2175             Grid[
2176               {
2177                 {template1[<|"numiter"->numiter, "totaliter"->
      totalIterations|>]},
2178                 {template2[<|"remtime"->Round[QuantityMagnitude[
      UnitConvert[(Now-startTime)/(Max[1,numiter])*(totalIterations-
      numiter), "min"]], 0.1]|>]},
2179                 {template4[<|"runtime"->Round[QuantityMagnitude[
      UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
2180                 {ProgressIndicator[numiter, {1, totalIterations}]}
2181               }
2182             ]
2183         ]
2184         ];
2185       )
2186     ];
2187     Do[
2188       (
2189         JJBlockMatrixTable[{numE, J, Jp}] = JJBlockMatrix[numE, J, Jp
      , CFTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas" ->
      OptionValue["ChenDeltas"]];
2190         numiter += 1;
2191       ),
2192     {Jp, AllowedJ[numE]},
2193     {J, AllowedJ[numE]}
2194     ];
```

```
    If[$FrontEnd =!= Null,
      NotebookDelete[temp]
    ];
    Return[JJBlockMatrixTable];
  );

  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'|>.";
  Options[TabulateManyJJBlockMatrixTables] = {"Overwrite"->False, "
    Sparse"->True, "ChenDeltas"->False, "FilenameAppendix"-> "", "
    Compressed" -> False};
  TabulateManyJJBlockMatrixTables[ns_, OptionsPattern[]]:= (
    overwrite = OptionValue["Overwrite"];
    fNames = <||>;
    fileApp = OptionValue["FilenameAppendix"];
    ExportFun = If[OptionValue["Compressed"], ExportMZip, Export];
    Do[
      (
        CFdataFilename = FileNameJoin[{moduleDir, "data", "
    CrystalFieldTable_f"<>ToString[numE]<>".zip"}];
        PrintTemporary["Importing CrystalFieldTable from ",
    CFdataFilename, " ..."];
        CrystalFieldTable = ImportMZip[CFdataFilename];

        PrintTemporary["#------- numE = ", numE, " -------#"];
        exportFname = JJBlockMatrixFileName[numE, "FilenameAppendix"
    -> fileApp];
        fNames[numE] = exportFname;
        If[FileExistsQ[exportFname] && Not[overwrite],
          Continue[]
        ];
        JJBlockMatrixTable = TabulateJJBlockMatrixTable[numE,
    CrystalFieldTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas"
    -> OptionValue["ChenDeltas"]];
        If[FileExistsQ[exportFname]&&overwrite,
          DeleteFile[exportFname]
        ];
        ExportFun[exportFname, JJBlockMatrixTable];

        ClearAll[CrystalFieldTable];
      ),
    {numE, ns}
    ];
    Return[fNames];
  );

  HamMatrixAssembly::usage="HamMatrixAssembly[numE] returns the
    Hamiltonian matrix for the f^n_i configuration. The matrix is
```

```
    returned as a SparseArray.
2234 The function admits an optional parameter \"FilenameAppendix\"
    which can be used to modify the filename to which the resulting
    array is exported to.
2235 It also admits an optional parameter \"IncludeZeeman\" which can be
     used to include the Zeeman interaction.
2236 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.
2237 The option \"ReturnInBlocks\" can be used to return the matrix in
    block or flattened form. The default is to return it in flattened
    form.";
2238 Options[HamMatrixAssembly] = {
2239       "FilenameAppendix"->"",
2240       "IncludeZeeman"->False,
2241       "Set t2Switch"->True,
2242       "ReturnInBlocks"->False};
2243 HamMatrixAssembly[nf_, OptionsPattern[]]:=Module[
2244   {numE, ii, jj, howManyJs, Js, blockHam},
2245   (
2246     (*######################################*)
2247     ImportFun = ImportMZip;
2248     (*######################################*)
2249     (*hole-particle equivalence enforcement*)
2250     numE = nf;
2251     allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2,
    T2p,
2252       T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
2253       α, β, γ, B02, B04, B06, B12, B14, B16,
2254       B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16,
    S22,
2255       S24, S26, S34, S36, S44, S46, S56, S66, T11, T11p, T12, T14,
    T15, T16,
2256       T17, T18, T19, Bx, By, Bz};
2257     params0 = AssociationThread[allVars, allVars];
2258     If[nf > 7,
2259       (
2260         numE = 14 - nf;
2261         params = HoleElectronConjugation[params0];
2262         If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
2263       ),
2264       params = params0;
2265       If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
2266     ];
2267     (* Load symbolic expressions for LS,J,J' energy sub-matrices.
    *)
2268     emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
    OptionValue["FilenameAppendix"]];
2269     JJBlockMatrixTable = ImportFun[emFname];
2270     (*Patch together the entire matrix representation using J,J'
    blocks.*)
2271     PrintTemporary["Patching JJ blocks ..."];
2272     Js        = AllowedJ[numE];
2273     howManyJs = Length[Js];
2274     blockHam  = ConstantArray[0, {howManyJs, howManyJs}];
```

```
2275        Do[
2276          blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[
      jj]]}];,
2277        {ii, 1, howManyJs},
2278        {jj, 1, howManyJs}
2279        ];
2280
2281        (* Once the block form is created flatten it *)
2282        If[Not[OptionValue["ReturnInBlocks"]],
2283          (blockHam = ArrayFlatten[blockHam];
2284          blockHam  = ReplaceInSparseArray[blockHam, params];
2285          ),
2286          (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
      ,{2}];)
2287        ];
2288
2289        If[OptionValue["IncludeZeeman"],
2290          (
2291            PrintTemporary["Including Zeeman terms ..."];
2292            {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
      ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
2293            blockHam += - teslaToKayser * (Bx * magx + By * magy + Bz *
       magz);
2294          )
2295        ];
2296        Return[blockHam];
2297      )
2298    ];
2299
2300    SimplerSymbolicHamMatrix::usage="SimplerSymbolicHamMatrix[numE,
      simplifier] is a simple addition to HamMatrixAssembly that applies
       a given simplification to the full Hamiltonian. simplifier is a
      list of replacement rules. If the option \"Export\" is set to True
      , then the function also exports the resulting sparse array to the
       ./hams/ folder. 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. The option \"
      IncludeZeeman\" can be used to toggle the inclusion of the Zeeman
      interaction with an external magnetic field.";
2301    Options[SimplerSymbolicHamMatrix]={
2302      "Export"->True,
2303      "PrependToFilename"->"",
2304      "EorF"->"F",
2305      "Overwrite" -> False,
2306      "Return" -> True,
2307      "Set t2Switch" -> False,
2308      "IncludeZeeman" -> False};
2309    SimplerSymbolicHamMatrix[numE_Integer, simplifier_List,
      OptionsPattern[]]:=Module[
2310      {thisHam,fname, fnamemx},
2311      (
2312        If[Not[ValueQ[ElectrostaticTable]],
2313          LoadElectrostatic[]
```

```
2314          ];
2315        If[Not[ValueQ[SOOandECSOTable]],
2316          LoadSOOandECSO[]
2317        ];
2318        If[Not[ValueQ[SpinOrbitTable]],
2319          LoadSpinOrbit[]
2320        ];
2321        If[Not[ValueQ[SpinSpinTable]],
2322          LoadSpinSpin[]
2323        ];
2324        If[Not[ValueQ[ThreeBodyTable]],
2325          LoadThreeBody[]
2326        ];
2327
2328        fname = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".m"}];
2329        fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".mx"}];
2330        If[Or[FileExistsQ[fname], FileExistsQ[fnamemx]] && Not[
    OptionValue["Overwrite"]],
2331          (
2332            If[OptionValue["Return"],
2333              (
2334                Which[
2335                  FileExistsQ[fnamemx],
2336                  (
2337                    Print["File ",fnamemx," already exists, and option
    \"Overwrite\" is set to False, loading file ..."];
2338                    thisHam = Import[fnamemx];
2339                    Return[thisHam];
2340                  ),
2341                  FileExistsQ[fname],
2342                  (
2343                    Print["File ",fname," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
2344                    thisHam = Import[fname];
2345                    Print["Exporting to file ",fnamemx, " for quicker
    loading."];
2346                    Export[fnamemx,thisHam];
2347                    Return[thisHam];
2348                  )
2349                ]
2350              ),
2351              (
2352                Print["File ",fname," already exists, skipping ..."];
2353                Return[Null];
2354              )
2355            ]
2356          )
2357        ];
2358
2359        thisHam = HamMatrixAssembly[numE, "Set t2Switch" -> OptionValue
    ["Set t2Switch"], "IncludeZeeman"->OptionValue["IncludeZeeman"]];
2360        thisHam = ReplaceInSparseArray[thisHam, simplifier];
2361        (* This removes zero entries from being included in the sparse
```

```
    array *)
2362      thisHam = SparseArray[thisHam];
2363      If[OptionValue["Export"],
2364      (
2365        Print["Exporting to file ",fname, " and to ", fnamemx];
2366        Export[fname,thisHam];
2367        Export[fnamemx,thisHam];
2368      )
2369      ];
2370      If[OptionValue["Return"],
2371        Return[thisHam],
2372        Return[Null]
2373      ];
2374    )
2375  ];
2376
2377  (* ###################### Block assembly ###################### *)
2378  (* ########################################################### *)
2379
2380  (* ########################################################### *)
2381  (* #################   Intermediate Coupling  ################# *)
2382
2383  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.";
2384  FreeHam[JJBlocks_List, numE_Integer]:=Module[
2385    {Js, basisJ, pivot, freeHam, idx, J, thisJbasis,
      shrunkBasisPositions, theBlock},
2386    (
2387      Js       = AllowedJ[numE];
2388      basisJ   = BasisLSJMJ[numE, "AsAssociation" -> True];
2389      pivot    = If[OddQ[numE], 1/2, 0];
2390      freeHam  = Table[(
2391        J           = Js[[idx]];
2392        theBlock    = JJBlocks[[idx]];
2393        thisJbasis  = basisJ[J];
2394        (* find the basis vectors that end with pivot *)
2395        shrunkBasisPositions = Flatten[Position[thisJbasis, {_ ...,
      pivot}]];
2396        (* take only those rows and columns *)
2397        theBlock[[shrunkBasisPositions, shrunkBasisPositions]]
2398      ),
2399      {idx, 1, Length[Js]}
2400      ];
2401      Return[freeHam];
2402    )
2403  ];
2404
2405  ListRepeater::usage="ListRepeater[list, reps] repeats each element
      of list reps times.";
2406  ListRepeater[list_List, repeats_Integer] := (
2407    Flatten[ConstantArray[#, repeats] & /@ list]
2408  );
2409
2410  ListLever::usage="ListLever[vecs, multiplicity] takes a list of
```

```
    vectors and returns all interleaved shifted versions of them.";
2411 ListLever[vecs_, multiplicity_]:=Module[
2412 {uppityVecs, uppityVec},
2413 (
2414   uppityVecs = Table[(
2415     uppityVec = PadRight[{#}, multiplicity] & /@ vec;
2416     uppityVec = Permutations /@ uppityVec;
2417     uppityVec = Transpose[uppityVec];
2418     uppityVec = Flatten /@ uppityVec
2419     ),
2420   {vec, vecs}
2421   ];
2422   Return[Flatten[uppityVecs, 1]];
2423 )
2424 ];
2425
2426 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.";
2427 EigenLever[eigenSys_, multiplicity_]:=Module[
2428   {eigenVals, eigenVecs, leveledEigenVecs, leveledEigenVals},
2429   (
2430     {eigenVals, eigenVecs} = eigenSys;
2431     leveledEigenVals       = ListRepeater[eigenVals, multiplicity];
2432     leveledEigenVecs       = ListLever[eigenVecs, multiplicity];
2433     Return[{Flatten[leveledEigenVals], leveledEigenVecs}]
2434   )
2435 ];
2436
2437
2438 SimplerSymbolicIntermediateHamMatrix::usage = "
    SimplerSymbolicIntermediateHamMatrix[numE] is provides a variation
     of HamMatrixAssembly that returns the intermediate Hamiltonian
    blocks applying a simplifier. 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.
2439 The option \"Simplifier\" is a list of symbols that are set to zero
     in the intermediate Hamiltoniann description. 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.
2440 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.
2441 The option \"PrependToFilename\" can be used to append a string to
    the filename to which the function may export to.
2442 The option \"Return\" can be used to choose whether the function
    returns the matrix or not.
2443 The option \"Overwrite\" can be used to overwrite the file if it
```

```
    already exists.";
2444  Options[SimplerSymbolicIntermediateHamMatrix] = {
2445    "Export" -> True,
2446    "PrependToFilename" -> "",
2447    "Overwrite" -> False,
2448    "Return" -> True,
2449    "Simplifier" -> Join[
2450      {F0,\[Sigma]SS},
2451      cfSymbols,
2452      TSymbols,
2453      casimirSymbols,
2454      pseudoMagneticSymbols,
2455      marvinSymbols,
2456      DeleteCases[magneticSymbols,ζ]
2457    ]
2458  };
2459  SimplerSymbolicIntermediateHamMatrix[numE_Integer, OptionsPattern
      []] := Module[
2460    {thisHamAssoc, Js, fname, fnamemx, hash, simplifier},
2461    (
2462      simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
2463      hash       = Hash[simplifier];
2464      If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
2465      If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
2466      If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
2467      If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
2468      If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
2469      fname   = FileNameJoin[{moduleDir,"hams",OptionValue["
      PrependToFilename"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
      numE]<>"-"<>ToString[hash]<>".m"}];
2470      fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
      PrependToFilename"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
      numE]<>"-"<>ToString[hash]<>".mx"}];
2471      If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue
      ["Overwrite"]],
2472        (
2473          If[OptionValue["Return"],
2474          (
2475            Which[FileExistsQ[fnamemx],
2476            (
2477              Print["File ",fnamemx," already exists, and option \"
      Overwrite\" is set to False, loading file ..."];
2478              thisHamAssoc=Import[fnamemx];
2479              Return[thisHamAssoc];
2480            ),
2481            FileExistsQ[fname],
2482            (
2483              Print["File ",fname," already exists, and option \"
      Overwrite\" is set to False, loading file ..."];
2484              thisHamAssoc=Import[fname];
2485              Print["Exporting to file ",fnamemx," for quicker loading.
      "];
2486              Export[fnamemx,thisHamAssoc];
2487              Return[thisHamAssoc];
2488            )
```

```
2489            ]
2490          ),
2491          (
2492            Print["File ",fname," already exists, skipping ..."];
2493            Return[Null];
2494          )
2495          ]
2496        )
2497        ];
2498        Js           = AllowedJ[numE];
2499        thisHamAssoc = HamMatrixAssembly[numE,
2500          "Set t2Switch"->True,
2501          "IncludeZeeman"->False,
2502          "ReturnInBlocks"->True
2503        ];
2504        thisHamAssoc = Diagonal[thisHamAssoc];
2505        thisHamAssoc = Map[SparseArray[ReplaceInSparseArray[#,
    simplifier]]&,thisHamAssoc,{1}];
2506        thisHamAssoc = FreeHam[thisHamAssoc,numE];
2507        thisHamAssoc = AssociationThread[Js->thisHamAssoc];
2508        If[OptionValue["Export"],
2509          (
2510            Print["Exporting to file ",fname," and to ",fnamemx];
2511            Export[fname,thisHamAssoc];
2512            Export[fnamemx,thisHamAssoc];
2513          )
2514        ];
2515        If[OptionValue["Return"],
2516          Return[thisHamAssoc],
2517          Return[Null]
2518        ];
2519      )
2520    ];
2521
2522    IntermediateSolver::usage="IntermediateSolver[numE, params] puts
      together (or retrieves from disk) the symbolic intermediate
      Hamiltonian for the f^numE configuration and solves it for the
      given params returning the resultant energies and eigenstates.
2523   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
      subpsace, 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.
2524   If the option \"Return as states\" is set to True, then the
      function returns a list with two elements. The first element is
      the global intermediate coupling basis for the f^numE
      configuration, given as a list of lists of the form {LS string, J
      }. The second 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. The energies given have been
```

```
    subtracted the energy of the ground state, and the states have
    been sorted in order of increasing energy.
2525  The following options are admitted:
2526  - \"Overwrite Hamiltonian\", if set to True the function will
    overwrite the symbolic Hamiltonian. Default is False.
2527  - \"Return as states\", see description above. Default is True.
2528  - \"Simplifier\", this is a list with symbols that are set to zero
    for defining the parameters kept in the intermediate coupling
    description.
2529  ";
2530  Options[IntermediateSolver] = {
2531    "Overwrite Hamiltonian" -> False,
2532    "Return as states" -> True,
2533    "Simplifier" -> Join[
2534      cfSymbols,
2535      TSymbols,
2536      casimirSymbols,
2537      pseudoMagneticSymbols,
2538      marvinSymbols,
2539      DeleteCases[magneticSymbols,ζ]
2540    ],
2541    "PrintFun" -> PrintTemporary
2542  };
2543  IntermediateSolver[numE_Integer, params0_Association,
    OptionsPattern[]] := Module[
2544    {ln, simplifier, simpleHam, basis, numHam, eigensys, startTime,
    endTime, diagonalTime, params=params0, globalBasis, eigenVectors,
    eigenEnergies, eigenJs, states, groundEnergy, allEnergies,
    PrintFun},
2545    (
2546      ln         = theLanthanides[[numE]];
2547      basis      = BasisLSJ[numE, "AsAssociation"->True];
2548      simplifier = OptionValue["Simplifier"];
2549      PrintFun   = OptionValue["PrintFun"];
2550      PrintFun["> IntermediateSolver for ",ln," with ",numE," f-
    electrons."];
2551      PrintFun["> Loading the symbolic intermediate coupling
    Hamiltonian ..."];
2552      simpleHam  = SimplerSymbolicIntermediateHamMatrix[numE,
2553        "Simplifier" -> simplifier,
2554        "Overwrite"  -> OptionValue["Overwrite Hamiltonian"]
2555      ];
2556      (* Everything that is not given is set to zero *)
2557      PrintFun["> Setting to zero every parameter not given ..."];
2558      params    = ParamPad[params,"Print"->True];
2559      PrintFun[params];
2560      (* Create the numeric hamiltonian *)
2561      PrintFun["> Replacing parameters in the J-blocks of the
    intermediate coupling Hamiltonian to produce numeric arrays ..."];
2562      numHam    = N /@ Map[ReplaceInSparseArray[#, params]&,
    simpleHam];
2563      Clear[simpleHam];
2564      (* Eigensolver *)
2565      PrintFun["> Diagonalizing the numerical Hamiltonian within each
     separat J-subspace ..."];
```

```
2566        startTime     = Now;
2567        eigensys      = Eigensystem /@ numHam;
2568        endTime       = Now;
2569        diagonalTime  = QuantityMagnitude[endTime-startTime,"Seconds"];
2570        allEnergies   = Flatten[First/@Values[eigensys]];
2571        groundEnergy  = Min[allEnergies];
2572        eigensys      = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys
      ];
2573        eigensys      = Association@KeyValueMap[#1->{basis[#1],#2} &,
      eigensys];
2574        PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
2575        If[OptionValue["Return as states"],
2576          (
2577            PrintFun["> Padding the eigenvectors to correspond to the
      global intermediate basis ..."];
2578            eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values
      [#[[2, 2]] & /@ eigensys]];
2579            globalBasis   = Flatten[Values[basis], 1];
2580            eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
2581            eigenJs       = Flatten[KeyValueMap[ConstantArray[#1,
      Length[#2[[2, 2]]]] &, eigensys]];
2582            states        = Transpose[{eigenEnergies, eigenJs,
      eigenVectors}];
2583            states        = SortBy[states, First];
2584            Return[{globalBasis, states}];
2585          ),
2586          Return[{basis, eigensys}]
2587        ];
2588      )
2589    ];
2590
2591
2592    (* ################   Intermediate Coupling  ################# *)
2593    (* ############################################################ *)
2594
2595    (* ############################################################ *)
2596    (* ################### Optical Operators ###################### *)
2597
2598    magOp = <||>;
2599
2600    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.
2601    See eqn 15.7 in TASS.
2602    Here it is provided in atomic units in which the Bohr magneton is
        1/2.
2603    \[Mu] = -(1/2) (L + gs S)
2604    We are using the Racah convention for the reduced matrix elements
        in the Wigner-Eckart theorem. See TASS eqn 11.15.
2605    ";
2606    Options[JJBlockMagDip]={"Sparse"->True};
2607    JJBlockMagDip[numE_, braJ_, ketJ_, OptionsPattern[]]:=Module[
2608      {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];
    )
  ];

  Options[TabulateJJBlockMagDipTable]={"Sparse"->True};
  TabulateJJBlockMagDipTable[numE_,OptionsPattern[]]:=(
    JJBlockMagDipTable=<||>;
    Js=AllowedJ[numE];
```

```
    Do[
      (
        JJBlockMagDipTable[{numE,braJ,ketJ}]=
          JJBlockMagDip[numE,braJ,ketJ,"Sparse"->OptionValue["Sparse"]]
      ),
    {braJ, Js},
    {ketJ, Js}
    ];
    Return[JJBlockMagDipTable]
  );

  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'|>.";
  Options[TabulateManyJJBlockMagDipTables]={"FilenameAppendix"->"","
    Overwrite"->False,"Compressed"->True};
  TabulateManyJJBlockMagDipTables[ns_,OptionsPattern[]]:=(
    fnames=<||>;
    Do[
    (
      ExportFun=If[OptionValue["Compressed"],ExportMZip,Export];
      PrintTemporary["#------- numE = ",numE," -------#"];
      appendTo     = (OptionValue["FilenameAppendix"]<>"-magDip");
      exportFname  = JJBlockMatrixFileName[numE,"FilenameAppendix"->
    appendTo];
      fnames[numE] = exportFname;
      If[FileExistsQ[exportFname]&&Not[OptionValue["Overwrite"]],
        Continue[]
      ];
      JJBlockMatrixTable = TabulateJJBlockMagDipTable[numE];
      If[FileExistsQ[exportFname]&&OptionValue["Overwrite"],
        DeleteFile[exportFname]
      ];
      ExportFun[exportFname,  JJBlockMatrixTable];
    ),
    {numE,ns}
    ];
    Return[fnames];
  );

  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.";
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}];
  )
];

MagDipLineStrength::usage="MagDipLineStrength[theEigensys, numE]
  takes the eigensystem of an ion and the number numE of f-electrons
   that correspond to it and it 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) or to \"Hartree\".
```

```
2745  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.
2746  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>.";
2747  Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI"
      , "States" -> All};
2748  MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern
      []]:=Module[
2749    {allEigenvecs, Sx, Sy, Sz, Stot ,factor},
2750    (
2751      numE = Min[14-numE0, numE0];
2752      (*If not loaded then load it, *)
2753      If[Or[
2754          Not[MemberQ[Keys[magOp], numE]],
2755          OptionValue["Reload MagOp"]],
2756      (
2757        magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
      MagDipoleMatrixAssembly[numE];
2758      )
2759      ];
2760      allEigenvecs = Transpose[Last /@ theEigensys];
2761      Which[OptionValue["States"] === All,
2762        (
2763          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
      allEigenvecs) & /@ magOp[numE];
2764          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2765        ),
2766        IntegerQ[OptionValue["States"]],
2767        (
2768          singleState  = theEigensys[[OptionValue["States"],2]];
2769          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
      singleState) & /@ magOp[numE];
2770          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2771        )
2772      ];
2773      Which[
2774        OptionValue["Units"] == "SI",
2775          Return[4 \[Mu]B^2 * Stot],
2776        OptionValue["Units"] == "Hartree",
2777          Return[Stot],
2778        True,
2779        (
2780          Print["Invalid option for \"Units\". Options are \"SI\" and
       \"Hartree\"."];
2781          Abort[];
2782        )
2783      ];
2784    )
2785  ];
2786
```

```
2787  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.
2788  Based on table 7.3 of Thorne 1999, using g2=1.
2789  The energy unit assumed in eigenSys is kayser.
2790  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>.
2791  By default this assumes that the refractive index is unity, this
        may be changed by setting the option \"RefractiveIndex\" to the
        desired value.
2792  The option \"Lifetime\" can be used to return the reciprocal of the
         transition rates. The default is to return the transition rates."
        ;
2793  Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False, "
        RefractiveIndex"->1};
2794  MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]]:=
        Module[
2795      {AMD,Stot,eigenEnergies,transitionWaveLengthsInMeters,nRefractive
        },
2796      (
2797        nRefractive    = OptionValue["RefractiveIndex"];
2798        numE           = Min[14-numE0, numE0];
2799        Stot           = MagDipLineStrength[eigenSys, numE, "Units"->
        OptionValue["Units"]];
2800        eigenEnergies  = Chop[First/@eigenSys];
2801        energyDiffs    = Outer[Subtract,eigenEnergies,eigenEnergies];
2802        energyDiffs    = ReplaceDiagonal[energyDiffs, Indeterminate];
2803        (* Energies assumed in pseudo-energy unit kayser.*)
2804        transitionWaveLengthsInMeters = 0.01/energyDiffs;
2805
2806        unitFactor     = Which[
2807        OptionValue["Units"]=="Hartree",
2808        (
2809          (* The bohrRadius factor in SI neede to convert the
        wavelengths which are assumed in m*)
2810          16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckFine)) * bohrRadius^3
2811        ),
2812        OptionValue["Units"]=="SI",
2813        (
2814          16 \[Pi]^3 \[Mu]0/(3 hPlanck)
2815        ),
2816        True,
2817        (
2818          Print["Invalid option for \"Units\". Options are \"SI\" and \
        "Hartree\"."];
2819          Abort[];
2820        )
2821        ];
```

```
2822        AMD = unitFactor / transitionWaveLengthsInMeters^3 * Stot *
      nRefractive^3;
2823        Which[OptionValue["Lifetime"],
2824          Return[1/AMD],
2825          True,
2826          Return[AMD]
2827        ]
2828      )
2829    ];
2830
2831    GroundMDOscillatorStrength::usage="GroundMDOscillatorStrength[
      eigenSys, numE] calculates the magnetic diople oscillator
      strengths between the ground state and the excited states as given
       by eigenSys.
2832    Based on equation 8 of Carnall 1965, removing the 2J+1 factor since
       this degeneracy has been removed by the crystal field.
2833    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.
2834    The energy unit assumed in eigenSys is Kayser.
2835    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>.
2836    By default this assumes that the refractive index is unity, this
      may be changed by setting the option \"RefractiveIndex\" to the
      desired value.";
2837    Options[GroundMDOscillatorStrength]={"RefractiveIndex"->1};
2838    GroundMDOscillatorStrength[eigenSys_List, numE_Integer,
      OptionsPattern[]]:=Module[
2839      {eigenEnergies, SMDGS, GSEnergy, energyDiffs,
      transitionWaveLengthsInMeters, unitFactor, nRefractive},
2840      (
2841        eigenEnergies     = First/@eigenSys;
2842        nRefractive       = OptionValue["RefractiveIndex"];
2843        SMDGS             = MagDipLineStrength[eigenSys,numE, "Units"->"
      SI", "States"->1];
2844        GSEnergy          = eigenSys[[1,1]];
2845        energyDiffs       = eigenEnergies-GSEnergy;
2846        energyDiffs[[1]] = Indeterminate;
2847        transitionWaveLengthsInMeters = 0.01/energyDiffs;
2848        unitFactor        = (8\[Pi]^2 me)/(3 hPlanck eCharge^2 cLight);
2849        fMDGS             = unitFactor / transitionWaveLengthsInMeters *
       SMDGS * nRefractive;
2850        Return[fMDGS];
2851      )
2852    ];
2853
2854    (* ################### Optical Operators ################### *)
2855    (* ########################################################## *)
2856
2857    (* ########################################################## *)
2858    (* ################### Printers and Labels ################## *)
2859
2860    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]]]]

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.";
AllowedSLJTerms[numE_] := Module[
  {idx1, allowedSL, allowedSLJ},
  (
    allowedSL = AllowedSLTerms[numE];
    allowedSLJ = {};
    For[
      idx1 = 1,
      idx1 <= Length[allowedSL],
      termSL = allowedSL[[idx1]];
      termsSLJ =
        Table[
          {termSL[[1]], termSL[[2]], J},
        {J,Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
        ];
      allowedSLJ = Join[allowedSLJ, termsSLJ];
      idx1++
    ];
    SortBy[allowedSLJ, Last]
  )
];

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.";
AllowedNKSLJTerms[numE_] := Module[
  {allowedSL, allowedNKSL, allowedSLJ, nn},
  (
    allowedNKSL = AllowedNKSLTerms[numE];
    allowedSL = AllowedSLTerms[numE];
    allowedSLJ = {};
    For[
      nn = 1,
      nn <= Length[allowedSL],
      (
        termSL = allowedSL[[nn]];
        termNKSL = allowedNKSL[[nn]];
```

```
2941            termsSLJ =
2942              Table[{termNKSL, J},
2943              {J, Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
2944              ];
2945            allowedSLJ = Join[allowedSLJ, termsSLJ];
2946            nn++
2947          )
2948        ];
2949        SortBy[allowedSLJ, Last]
2950      )
2951    ];
2952
2953    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.";
2954    AllowedNKSLforJTerms[numE_, J_]:=Module[
2955      {allowedSL, allowedNKSL, allowedSLJ, nn, termSL, termNKSL,
       termsSLJ},
2956      (
2957        allowedNKSL = AllowedNKSLTerms[numE];
2958        allowedSL = AllowedSLTerms[numE];
2959        allowedSLJ = {};
2960        For[
2961          nn = 1,
2962          nn <= Length[allowedSL],
2963          (
2964            termSL = allowedSL[[nn]];
2965            termNKSL = allowedNKSL[[nn]];
2966            termsSLJ = If[Abs[termSL[[1]] - termSL[[2]]] <= J <= Total[
       termSL],
2967              {{termNKSL, J}},
2968              {}
2969              ];
2970            allowedSLJ = Join[allowedSLJ, termsSLJ];
2971            nn++
2972          )
2973        ];
2974        Return[allowedSLJ]
2975      )
2976    ];
2977
2978    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
       }.";
2979    AllowedSLJMTerms[numE_]:=Module[
2980      {allowedSLJ, allowedSLJM, termSLJ, termsSLJM, nn},
2981      (
2982      allowedSLJ = AllowedSLJTerms[numE];
2983      allowedSLJM = {};
2984      For[
2985        nn = 1,
2986        nn <= Length[allowedSLJ],
```

```
2987        nn++,
2988        (
2989          termSLJ = allowedSLJ[[nn]];
2990          termsSLJM =
2991            Table[{termSLJ[[1]], termSLJ[[2]], termSLJ[[3]], M},
2992            {M, - termSLJ[[3]], termSLJ[[3]]}
2993            ];
2994          allowedSLJM = Join[allowedSLJM, termsSLJM];
2995        )
2996      ];
2997      Return[SortBy[allowedSLJM, Last]];
2998      )
2999    ];
3000
3001    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}.";
3002    AllowedNKSLJMforJMTerms[numE_, J_, MJ_] := Module[
3003      {allowedSL, allowedNKSL, allowedSLJM, nn},
3004      (
3005        allowedNKSL = AllowedNKSLTerms[numE];
3006        allowedSL   = AllowedSLTerms[numE];
3007        allowedSLJM = {};
3008        For[
3009          nn = 1,
3010          nn <= Length[allowedSL],
3011          termSL = allowedSL[[nn]];
3012          termNKSL = allowedNKSL[[nn]];
3013          termsSLJ = If[(Abs[termSL[[1]] - termSL[[2]]]
3014                         <= J
3015                         <= Total[termSL]
3016                         && (Abs[MJ] <= J)
3017                         ),
3018                         {{termNKSL, J, MJ}},
3019                         {}];
3020          allowedSLJM = Join[allowedSLJM, termsSLJ];
3021          nn++
3022        ];
3023        Return[allowedSLJM];
3024      )
3025    ];
3026
3027    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}, and if the option \"
       Flat\" is set to True then the returned list has element of the
       form {SL (string in spectroscopic notation), J, MJ}.";
3028    AllowedNKSLJMforJTerms[numE_, J_] := Module[
3029      {MJs, labelsAndMomenta, termsWithJ},
3030      (
3031        MJs = AllowedMforJ[J];
3032        (* Pair LS labels and their {S,L} momenta *)
```

```
3033        labelsAndMomenta = ({#, FindSL[#]}) & /@ AllowedNKSLTerms[numE
      ];
3034        (* A given term will contain J if |L-S|<=J<=L+S *)
3035        ContainsJ[{SL_String, {S_, L_}}] := (Abs[S - L] <= J <= (S + L)
      );
3036        (* Keep just the terms that satisfy this condition *)
3037        termsWithJ = Select[labelsAndMomenta, ContainsJ];
3038        (* We don't want to keep the {S,L} *)
3039        termsWithJ = {#[[1]], J} & /@ termsWithJ;
3040        (* This is just a quick way of including up all the MJ values
      *)
3041        Return[Flatten /@ Tuples[{termsWithJ, MJs}]]
3042      )
3043    ];
3044
3045    AllowedMforJ::usage = "AllowedMforJ[J] is shorthand for Range[-J, J
      , 1].";
3046    AllowedMforJ[J_] := Range[-J, J, 1];
3047
3048    AllowedJ::usage = "AllowedJ[numE] returns the total angular momenta
       J that appear in the f^numE configuration.";
3049    AllowedJ[numE_] := Table[J, {J, MinJ[numE], MaxJ[numE]}];
3050
3051    Seniority::usage="Seniority[LS] returns the seniority of the given
      term.";
3052    Seniority[LS_] := FindNKLSTerm[LS][[1, 2]];
3053
3054    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}.";
3055    FindNKLSTerm[SL_]:=Module[
3056      {NKterms, n},
3057      (
3058        n = 7;
3059        NKterms = {{}};
3060        Map[
3061          If[! StringFreeQ[First[#], SL],
3062            If[ToExpression[Part[#, 2]] <= n,
3063              NKterms = Join[NKterms, {#}, 1]
3064            ]
3065          ] &,
3066        fnTermLabels
3067        ];
3068        NKterms = DeleteCases[NKterms, {}];
3069        NKterms
3070      )
3071    ];
3072
3073    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}.";
3074    Options[ParseTermLabels] = {"Export" -> True};
```

```
ParseTermLabels[OptionsPattern[]]:=Module[
  {labelsTextData, fNtextLabels, nielsonKosterLabels, seniorities,
  RacahW, RacahU},
  (
    labelsTextData = FileNameJoin[{moduleDir, "data", "
  NielsonKosterLabels_f6_f7.txt"}];
    fNtextLabels   = Import[labelsTextData];
    nielsonKosterLabels = Partition[StringSplit[fNtextLabels], 3];
    termLabels = Map[Part[#, {1}] &, nielsonKosterLabels];
    seniorities = Map[ToExpression[Part[# , {2}]] &,
  nielsonKosterLabels];
    racahW =
      Map[
        StringTake[
          Flatten[StringCases[Part[# , {3}],
            "(" ~~ DigitCharacter ~~ DigitCharacter ~~
  DigitCharacter ~~ ")"]],
          {2, 4}
        ] &,
      nielsonKosterLabels];
    racahU =
      Map[
        StringTake[
          Flatten[StringCases[Part[# , {3}],
            "(" ~~ DigitCharacter ~~ DigitCharacter ~~ ")"]],
          {2, 3}
        ] &,
      nielsonKosterLabels];
    fnTermLabels = Join[termLabels, seniorities, racahW, racahU,
  2];
    fnTermLabels = Sort[fnTermLabels];
    If[OptionValue["Export"],
      (
        broadFname = FileNameJoin[{moduleDir,"data","fnTerms.m"}];
        Export[broadFname, fnTermLabels];
      )
    ];
    Return[fnTermLabels];
  )
];

(* ###################### Term management ###################### *)
(* ############################################################## *)

LoadParameters::usage="LoadParameters[ln] takes a string with the
  symbol the element of a trivalent lanthanide ion and returns model
   parameters for it. It is based on the data for LaF3. If the
  option \"Free Ion\" is set to True then the function sets all
  crystal field parameters to zero. Through the option \"gs\" it
  allows modyfing the electronic gyromagnetic ratio. For
  completeness this function also computes the E parameters using
  the F parameters quoted on Carnall.";
Options[LoadParameters] = {
    "Source"->"Carnall",
    "Free Ion"->False,
```

```
3118        "gs"->2.002319304386,
3119        "With Uncertainties"->False
3120        };
3121    LoadParameters[Ln_String, OptionsPattern[]]:= Module[
3122      {
3123        source, params, uncertain, uncertainKeys, uncertainRules
3124      },
3125      (
3126        If[Not[ValueQ[Carnall]],
3127          LoadCarnall[];
3128        ];
3129        source = OptionValue["Source"];
3130        params = Which[source=="Carnall",
3131                 (Association[Carnall["data"][Ln]])
3132                 ];
3133        (*If a free ion then all the parameters from the crystal field
      are set to zero*)
3134        If[OptionValue["Free Ion"],
3135          Do[params[cfSymbol] = 0, {cfSymbol, cfSymbols}]
3136        ];
3137        params[F0] = 0;
3138        params[M2] = 0.56*params[M0]; (*See Carnall 1989,Table I,
      caption,probably fixed based on HF values*)
3139        params[M4] = 0.31*params[M0]; (*See Carnall 1989,Table I,
      caption,probably fixed based on HF values*)
3140        params[P0] = 0;
3141        params[P4] = 0.5*params[P2];  (*See Carnall 1989,Table I,
      caption,probably fixed based on HF values*)
3142        params[P6] = 0.1*params[P2];  (*See Carnall 1989,Table I,
      caption,probably fixed based on HF values*)
3143        params[gs] = OptionValue["gs"];
3144        {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[
      F0], params[F2], params[F4], params[F6]];
3145        params[E0] = 0;
3146        If[
3147          Not[OptionValue["With Uncertainties"]],
3148          Return[params],
3149          (
3150            uncertain     = Association[Carnall["annotations"][Ln]];
3151            uncertainKeys = Keys[uncertain];
3152            uncertain     = If[# == "Not allowed to vary in fitting."
      || # == "Interpolated",
3153                0., #] & /@ uncertain;
3154            paramKeys = Keys[params];
3155            uncertainVals = Sort[Intersection[paramKeys, uncertainKeys
      ]] /. Association[uncertain];
3156            uncertainRules = MapThread[Rule, {Sort[uncertainKeys],
      uncertainVals}];
3157           Which[
3158             MemberQ[{"Ce", "Yb"}, Ln],
3159             (
3160               subsetL = {F0};
3161               subsetR = {0};
3162             ),
3163             True,
```

```
3164            (
3165            subsetL = {F0, M2, M4, P0, P4, P6, E0, E1, E2, E3};
3166            subsetR = {0, M0*0.65, M0*0.31, 0, P2*0.5, P2*0.1,
3167              0,
3168              Sqrt[(196 F2^2)/164025 + (49 F4^2)/88209 + (122500 F6
    ^2)/134165889],
3169              Sqrt[F2^2/4100625 + F4^2/10673289 + (30625 F6^2)
    /2743558264161],
3170              Sqrt[F2^2/18225 + (4 F4^2)/1185921 + (30625 F6^2)
    /1803785841]};
3171            )
3172          ];
3173          uncertainRules = Join[uncertainRules, MapThread[Rule, {
    subsetL,subsetR /. uncertainRules}]];
3174          uncertainRules = Association[uncertainRules];
3175          Which[
3176            Ln == "Eu",
3177            (
3178              uncertainRules[F4] = 12.121;
3179              uncertainRules[F6] = 15.872;
3180            ),
3181            Ln == "Gd",
3182            (
3183              uncertainRules[F4] = 12.07;
3184            ),
3185            Ln == "Tb",
3186            (
3187              uncertainRules[F4] = 41.006;
3188            )
3189          ];
3190          If[MemberQ[{"Eu","Gd","Tb"},Ln],
3191            (
3192              uncertainRules[E1] = Sqrt[(196 F2^2)/164025 + (49 F4^2)
    /88209 + (122500 F6^2)/134165889] /. uncertainRules;
3193              uncertainRules[E2] = Sqrt[F2^2/4100625 + F4^2/10673289
    + (30625 F6^2)/2743558264161] /. uncertainRules;
3194              uncertainRules[E3] = Sqrt[F2^2/18225 + (4 F4^2)/1185921
     + (30625 F6^2)/1803785841] /. uncertainRules;
3195            )
3196          ];
3197          uncertainKeys = First /@ Normal[uncertainRules];
3198          fullParams = Association[MapThread[Rule, {uncertainKeys,
    MapThread[Around, {uncertainKeys /. params, uncertainKeys /.
    uncertainRules}]}]];
3199          Return[Join[params, fullParams]]
3200        )
3201        ];
3202    )
3203  ];
3204
3205  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 complentary 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 the
    relationship is more complex and is controlled by the value of the
     isE variable.";
3206  HoleElectronConjugation[params_] := Module[
3207    {newparams = params},
3208    (
3209      flipSignsOf = {ζ, T2, T3, T4, T6, T7, T8};
3210      flipSignsOf = Join[flipSignsOf, cfSymbols];
3211      flipped =
3212        Table[(flipper -> - newparams[flipper]),
3213        {flipper, flipSignsOf}
3214        ];
3215      nonflipped =
3216        Table[(flipper -> newparams[flipper]),
3217        {flipper, Complement[Keys[newparams], flipSignsOf]}
3218        ];
3219      flippedParams = Association[Join[nonflipped, flipped]];
3220      flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
3221      Return[flippedParams];
3222    )
3223  ];
3224
3225  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.
3226  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.
3227  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\".
3228  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.
3229  Tha ordered basis in which this eigenvector is to be interpreted is
       the one corresponding to BasisLSJMJ[numE].
3230  The function admits the following options:
3231  \"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.
3232  \"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.
3233  \"Zeroes\" (list) : A list with symbols assumed to be zero.
3234  ";
3235  Options[IonSolver] = {
3236    "Include Spin-Spin" -> True,
3237    "Overwrite Hamiltonian" -> False,
3238    "Zeroes" -> {}
3239  };
3240  IonSolver[numE_Integer, params0_Association, host_String:"Ln",
```

```
    OptionsPattern[]]:=Module[
3241    {ln, simplifier, simpleHam, numHam, eigensys,
3242    startTime, endTime, diagonalTime, params=params0, zeroSymbols},
3243    (
3244      ln = theLanthanides[[numE]];
3245
3246      (* This could be done when replacing values, but this produces
        smaller saved arrays. *)
3247      simplifier = (#-> 0) & /@ OptionValue["Zeroes"];
3248      simpleHam = SimplerSymbolicHamMatrix[numE,
3249        simplifier,
3250        "PrependToFilename" -> host,
3251        "Overwrite" -> OptionValue["Overwrite Hamiltonian"]
3252      ];
3253
3254      (* Note that we don't have to flip signs of parameters for fn
        beyond f7 since the matrix produced
3255      by SimplerSymbolicHamMatrix has already accounted for this. *)
3256
3257      (* Everything that is not given is set to zero *)
3258      params = ParamPad[params, "Print" -> True];
3259      PrintFun[params];
3260
3261      (* Enforce the override to the spin-spin contribution to the
        magnetic interactions *)
3262      params[\[Sigma]SS] = If[OptionValue["Include Spin-Spin"], 1,
        0];
3263
3264      (* Create the numeric hamiltonian *)
3265      numHam = ReplaceInSparseArray[simpleHam, params];
3266      Clear[simpleHam];
3267
3268      (* Eigensolver *)
3269      PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
3270      startTime = Now;
3271      eigensys  = Eigensystem[numHam];
3272      endTime   = Now;
3273      diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"
        ];
3274      PrintFun[">> Diagonalization took ", diagonalTime, " seconds."
        ];
3275      eigensys = Chop[eigensys];
3276      eigensys = Transpose[eigensys];
3277
3278      (* Shift the baseline energy *)
3279      eigensys = ShiftedLevels[eigensys];
3280      (* Sort according to energy *)
3281      eigensys = SortBy[eigensys, First];
3282      Return[eigensys];
3283    )
3284  ];
3285
3286  ShiftedLevels::usage = "ShiftedLevels[eigenSys] takes a list of
      levels of the form
3287  {{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.";
3288 ShiftedLevels[originalLevels_] := Module[
3289   {groundEnergy, shifted},
3290   (
3291     groundEnergy = Sort[originalLevels][[1,1]];
3292     shifted      = Map[{#[[1]] - groundEnergy, #[[2]]} &,
     originalLevels];
3293     Return[shifted];
3294   )
3295 ];
3296
3297
3298 (* ############################################################ *)
3299 (* ###### Optical Transitions in Intermediate Coupling ####### *)
3300
3301 JuddOfeltUkSquared::usage="JuddOfeltUkSquared[numE, params]
     calculates the matrix elements of the Uk operator in the
     intermediate eigenstate basis. These are calculated according to
     equation (7) in Carnall 1965.
3302 The function returns a list with the following elements:
3303   - basis : A list with the allowed {SL, J} terms in the f^numE
     configuration. Equal to BasisLSJ[numE].
3304   - eigenSys : A list with the eigensystem of the Hamiltonian for
     the f^n configuration.
3305   - levelLabels : A list with the labels of the major components of
      the intermediate coupling eigenstates.
3306   - UkintermediateSquared : An association with the squared matrix
     elements of the Uk operators in the intermediate eigenstate basis.
      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 intermediate coupling eigenstates given
     in eigenSys and whose major SLJ components are given in
     levelLabels. The matrix is symmetric and given as a
     SymmetrizedArray.
3307   The function admits the following options:
3308   \"PrintFun\" : A function that will be used to print the progress
      of the calculations. The default is PrintTemporary.";
3309 Options[JuddOfeltUkSquared] = {"PrintFun" -> PrintTemporary};
3310 JuddOfeltUkSquared[numE_, params_, OptionsPattern[]] := Module[
3311   {eigenChanger, numEH, basis, eigenSys, Js, Ukmat,
     UkintermediateSquared, kRank, S, L, Sp, Lp, J, Jp, phase, braTerm,
      ketTerm, levelLabels, eigenVecs, majorComponentIndices},
3312   (
3313     If[Not[ValueQ[ReducedUkTable]],
3314       LoadUk[]
3315     ];
3316     numEH = Min[numE, 14-numE];
3317     PrintFun = OptionValue["PrintFun"];
3318     PrintFun["> Calculating the intermediate levels for the given
     parameters ..."];
3319     {basis, eigenSys} = IntermediateSolver[numE, params];
3320     (* The change of basis matrix to the eigenstate basis *)
3321     eigenChanger = Transpose[Last /@ eigenSys];
3322     PrintFun["Calculating the matrix elements of Uk in the physical
```

```
    coupling basis ..."];
      UkintermediateSquared = <||>;
      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;
        UkintermediateSquared[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, UkintermediateSquared}];
    )
  ];

  IntermediateOscillatorStrengthED::usage="
    IntermediateOscillatorStrengthED[numE, intermediateParams, juddOfeltParams] uses Judd-Ofelt theory to estimate the forced electric dipole oscillator strengths ions whose intermediate coupling description is determined by intermediateParams.
  The third parameter juddOfeltParams is an asssociation 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 intermediate coupling.
    - levelLabels : A list with the labels of the major components of the intermediate coupling levels.
    - oStrengthArray : A square arrayt whose elements represent the
```

```
    oscillator strengths between the levels of the intermediate
    coupling eigenstates 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.
3363  The function admits the following three options:
3364    \"PrintFun\" : A function that will be used to print the progress
     of the calculations. The default is PrintTemporary.
3365    \"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.
3366    \"LocalFieldCorrection\" : The local field correction to be used.
     The default is \"VirtualCavity\". The options are: \"
    VirtualCavity\" and \"EmptyCavity\".
3367  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.
3368  ";
3369  Options[IntermediateOscillatorStrengthED]={
3370    "PrintFun"        -> PrintTemporary,
3371    "RefractiveIndex" -> 1,
3372    "LocalFieldCorrection" -> "VirtualCavity"
3373  };
3374  IntermediateOscillatorStrengthED[numE_,
    intermediateParams_Association, juddOfeltParams_Association,
    OptionsPattern[]] := Module[
3375    {
3376      PrintFun, basis, eigenSys, levelLabels, UkintermediateSquared,
    eigenEnergies, energyDiffs, oStrengthArray, nRef, \[Chi], nRefs,
    \[Chi]OverN, groundLevel, const, transitionFrequencies,
    wavelengthsInNM, fieldCorrectionType
3377    },
3378    (
3379      PrintFun = OptionValue["PrintFun"];
3380      nRef     = OptionValue["RefractiveIndex"];
3381      PrintFun["Calculating the Uk^2 matrix elements for the given
    parameters ..."];
3382      {basis, eigenSys, levelLabels, UkintermediateSquared} =
    JuddOfeltUkSquared[numE, intermediateParams, "PrintFun"->PrintFun
    ];
3383      eigenEnergies = First/@eigenSys;
3384      const         = (8\[Pi]^2)/3 me/hPlanck;
3385      energyDiffs   = Transpose@Outer[Subtract,eigenEnergies,
    eigenEnergies];
3386      (* since energies are assumed in Kayser, speed of light needs
    to be in cm/s, so that the frequencies are in 1/s *)
3387      transitionFrequencies = energyDiffs*cLight*100;
3388      (* grab the J for each level *)
3389      levelJs        = #[[2]] & /@ eigenSys;
3390      oStrengthArray = (
```

```
3391        juddOfeltParams[\[CapitalOmega]2]*UkintermediateSquared[2]+
3392        juddOfeltParams[\[CapitalOmega]4]*UkintermediateSquared[4]+
3393        juddOfeltParams[\[CapitalOmega]6]*UkintermediateSquared[6]
3394      );
3395      oStrengthArray = Abs@(const * transitionFrequencies *
    oStrengthArray);
3396      (* it is necessary to divide each oscillator strength by the
    degeneracy of the initial level *)
3397      oStrengthArray = MapIndexed[1/(2 levelJs[[#2[[1]]]]+1) #1 &,
    oStrengthArray,{2}];
3398      (* including the effects of the refractive index *)
3399      fieldCorrectionType = OptionValue["LocalFieldCorrection"];
3400      Which[
3401        nRef === 1,
3402          True,
3403        NumberQ[nRef],
3404        (
3405          \[Chi] = Which[
3406            fieldCorrectionType == "VirtualCavity",
3407            (
3408              ( (nRef^2 + 2) / 3 )^2
3409            ),
3410            fieldCorrectionType == "EmptyCavity",
3411            (
3412              ( 3 * nRef^2 / ( 2 * nRef^2 + 1 ) )^2
3413            )
3414          ];
3415          \[Chi]OverN    = \[Chi] / nRef;
3416          oStrengthArray = \[Chi]OverN * oStrengthArray;
3417          (* the refractive index participates different in
    absorption and in emission *)
3418          aFunction      = If[#2[[1]]>#2[[2]], #1 * nRef^2, #1]&;
3419          oStrengthArray = MapIndexed[aFunction, oStrengthArray,
    {2}];
3420        ),
3421        True,
3422        (
3423          wavelengthsInNM = Abs[1 / energyDiffs] * 10^7;
3424          nRefs           = Map[nRef, wavelengthsInNM];
3425          Print["Calculating the oscillator strengths for the given
    refractive index ..."];
3426          \[Chi] = Which[
3427            fieldCorrectionType == "VirtualCavity",
3428            (
3429              ( (nRefs^2 + 2) / 3 )^2
3430            ),
3431            fieldCorrectionType == "EmptyCavity",
3432            (
3433              ( 3 * nRefs^2 / ( 2*nRefs^2 + 1 ) )^2
3434            )
3435          ];
3436          \[Chi]OverN     = \[Chi] / nRefs;
3437          oStrengthArray  = \[Chi]OverN * oStrengthArray
3438        )
3439      ];
```

```
3440        Return[{basis, eigenSys, levelLabels, oStrengthArray}];
3441      )
3442    ];
3443
3444    JJBlockMagDipIntermediate::usage="JJBlockMagDipIntermediate[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.";
3445    Options[JJBlockMagDipIntermediate] = {"Sparse"->True};
3446    JJBlockMagDipIntermediate[numE_, braJ_, ketJ_, OptionsPattern[]] :=
         Module[
3447      {
3448        braSLJs, ketSLJs, braSLJ, ketSLJ, braSL, ketSL, braS, braL,
        ketS, ketL, matValue, magMatrix, summand1, summand2
3449      },
3450    (
3451      braSLJs     = AllowedNKSLforJTerms[numE,braJ];
3452      ketSLJs     = AllowedNKSLforJTerms[numE,ketJ];
3453      magMatrix   = Table[
3454        braSL       = braSLJ[[1]];
3455        ketSL       = ketSLJ[[1]];
3456        {braS,braL} = FindSL[braSL];
3457        {ketS,ketL} = FindSL[ketSL];
3458        summand1    = If[Or[braJ!=ketJ,braSL!=ketSL],
3459          0,
3460          Sqrt[braJ*(braJ+1)*TPO[braJ]
3461          ]
3462        ];
3463        (*looking at the string includes checking L=L',S=S',and\alpha=\
        alpha'*)summand2    = If[braSL!=ketSL,
3464          0,
3465          (gs-1)*
3466          Phaser[braS+braL+ketJ+1]*
3467          Sqrt[TPO[braJ]*TPO[ketJ]]*
3468          SixJay[{braJ,1,ketJ},{braS,braL,braS}]*
3469          Sqrt[braS(braS+1)TPO[braS]]
3470        ];
3471        matValue = summand1+summand2;
3472        matValue = -1/2*matValue;
3473        matValue,
3474        {braSLJ,braSLJs},
3475        {ketSLJ,ketSLJs}
3476      ];
3477      If[OptionValue["Sparse"],
3478        magMatrix=SparseArray[magMatrix]];
3479      Return[magMatrix];
3480    )
3481    ];
3482
3483    IntermediateMagDipole::usage="IntermediateMagDipole[numE] puts
          together an array with the reduced matrix elements of the magnetic
           dipole operator in the intermediate coupling basis for the f^numE
           configuration. The function admits the two following options:
          \"Flattened\" (bool) : If True then the returned matrix is
```

```
  flattened. The default is True.
3485  \"gs\" (number) : The electronic gyromagnetic ratio. The default
  is 2.";
3486 Options[IntermediateMagDipole] = {"Flattened" -> True, gs -> 2};
3487 IntermediateMagDipole[numE_, OptionsPattern[]] := Module[
3488   {
3489     Js, magDip, braJ, ketJ
3490   },
3491   (
3492     Js      = AllowedJ[numE];
3493     magDip  = Table[
3494       ReplaceInSparseArray[
3495         JJBlockMagDipIntermediate[numE,braJ,ketJ],
3496         {gs->OptionValue[gs]}
3497         ],
3498       {braJ,Js},
3499       {ketJ,Js}
3500     ];
3501     If[OptionValue["Flattened"],
3502       magDip = ArrayFlatten[magDip];
3503     ];
3504     Return[magDip];
3505   )
3506 ];
3507
3508
3509 IntermediateOscillatorStrengthMD::usage = "
  IntermediateOscillatorStrengthMD[numE, intermediateParams] uses
  Judd-Ofelt theory to estimate the forced electric dipole
  oscillator strengths for an ion whose intermediate coupling
  description is determined by intermediateParams.
3510 The function returns a square array, oStrengthArray, where
  oStrengthArray[[i,j]] equals the oscillator strength (which is a
  dimensionless quantity) between levels |Subscript[\[Psi], i]> and
  |Subscript[\[Psi], j]>.
3511 The function returns a list with the following elements:
3512   - basis : A list with the allowed {SL, J} terms in the f^numE
  configuration. Equal to BasisLSJ[numE].
3513   - eigenSys : A list with the eigensystem of the Hamiltonian for
  the f^n configuration in intermediate coupling.
3514   - levelLabels : A list with the labels of the major components of
   the intermediate coupling levels.
3515   - magDipoleOstrength : A square array whose elements represent
  the magnetic dipole oscillator strengths between the levels of the
   intermediate coupling eigenstates 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.
3516 The function admits the following two options:
3517   \"PrintFun\" : A function that will be used to print the progress
   of the calculations. The default is PrintTemporary.
3518   \"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.
3519 ";
3520 Options[IntermediateOscillatorStrengthMD] = {
3521   "PrintFun" -> PrintTemporary
3522 };
3523 IntermediateOscillatorStrengthMD[numE_,
    intermediateParams_Association,  OptionsPattern[]] := Module[
3524   {
3525     PrintFun, eigenSys, eigenEnergies, interLevels, energyDiffs,
    levelJs, magDipoleOstrength, LSJmultiplets, majorComponentIndices,
     levelLabels
3526   },
3527   (
3528     PrintFun = OptionValue["PrintFun"];
3529     PrintFun["> Calculating the intermediate levels for the given
    parameters ..."];
3530     {basis, eigenSys} = IntermediateSolver[numE, intermediateParams
    ];
3531     (* The change of basis matrix to the eigenstate basis *)
3532     eigenEnergies  = First /@ eigenSys;
3533     interLevels    = Transpose[Last /@ eigenSys];
3534     energyDiffs    = Abs@Outer[Subtract, eigenEnergies,
    eigenEnergies];
3535     energyDiffs   *= kayserToHartree;
3536     levelJs        = #[[2]] & /@ eigenSys;
3537     magDip         = IntermediateMagDipole[numE];
3538     magDipoleOstrength = (2/3 *
3539       αFine^2
3540       energyDiffs *
3541       Abs[Transpose[interLevels] . magDip . interLevels]^2
3542     );
3543     magDipoleOstrength = MapIndexed[
3544       1/(2 * levelJs[[#2[[1]]]] + 1) * #1 &,
3545       magDipoleOstrength,
3546       {2}
3547     ];
3548     LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
    InputForm[#[[2]]]]) & /@ basis;
3549     majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ Transpose[
    interLevels];
3550     levelLabels = LSJmultiplets[[majorComponentIndices]];
3551     Return[{basis, eigenSys, levelLabels, magDipoleOstrength}];
3552   )
3553 ];
3554
3555
3556 (* ###### Optical Transitions in Intermediate Coupling ####### *)
3557 (* ############################################################ *)
3558
3559 (* ############################################################ *)
3560 (* ################### Eigensystem analysis ################### *)
```

```
3561
3562  PrettySaundersSLJmJ::usage = "PrettySaundersSLJmJ[{SL, J, mJ}]
        produces a human-redeable symbol for the given basis vector {SL, J
        , mJ}.";
3563  Options[PrettySaundersSLJmJ] = {"Representation" -> "Ket"};
3564  PrettySaundersSLJmJ[{SL_, J_, mJ_}, OptionsPattern[]] := (If[
3565    StringQ[SL],
3566    ({S, L} = FindSL[SL];
3567      L = StringTake[SL, {2, -1}];
3568      ),
3569    {S, L} = SL];
3570    pretty = RowBox[{AdjustmentBox[Style[2*S + 1, Smaller],
3571        BoxBaselineShift -> -1, BoxMargins -> 0],
3572        AdjustmentBox[PrintL[L], BoxMargins -> -0.2],
3573        AdjustmentBox[
3574        Style[{InputForm[J], mJ}, Small, FontTracking -> "Narrow"],
3575        BoxBaselineShift -> 1,
3576        BoxMargins -> {{0.7, 0}, {0.4, 0.4}}]}];
3577    pretty = DisplayForm[pretty];
3578    If[OptionValue["Representation"] == "Ket",
3579      pretty = Ket[pretty]
3580    ];
3581    Return[pretty];
3582  );
3583
3584  PrettySaundersSLJ::usage = "PrettySaundersSLJ[{SL, J}] produces a
        human-redeable symbol for the given basis vector {SL, J}. SL can
        be either a list of two numbers representing S and L or a string
        representing the spin multiplicity and the total orbital angular
        momentum J in spectroscopic notation. The option \"Representation\
        " can be used to specify whether the output is given as a symbol
        or as a ket. The default is \"Ket\".";
3585  Options[PrettySaundersSLJ] = {"Representation"->"Ket"};
3586  PrettySaundersSLJ[{SL_,J_},OptionsPattern[]]:=(
3587    If[StringQ[SL],
3588    (
3589      {S,L}=FindSL[SL];
3590      L=StringTake[SL,{2,-1}];
3591    ),
3592      {S,L}=SL
3593    ];
3594    pretty = RowBox[{
3595        AdjustmentBox[Style[2*S+1,Smaller],BoxBaselineShift->-1,
      BoxMargins->0],
3596        AdjustmentBox[PrintL[L],BoxMargins->-0.2],
3597        AdjustmentBox[Style[InputForm[J],Small,FontTracking->"Narrow"
      ],BoxBaselineShift->1,BoxMargins->{{0.7,0},{0.4,0.4}}]
3598      }
3599    ];
3600    pretty = DisplayForm[pretty];
3601    pretty = Which[
3602      OptionValue["Representation"]=="Ket",
3603        Ket[pretty],
3604      OptionValue["Representation"]=="Symbol",
3605        pretty
```

```
3606      ];
3607      Return[pretty];
3608    );
3609
3610    BasisVecInRusselSaunders::usage = "BasisVecInRusselSaunders[
            basisVec] takes a basis vector in the format {LSstring, Jval,
            mJval} and returns a human-readable symbol for the corresponding
            Russel-Saunders term."
3611    BasisVecInRusselSaunders[basisVec_] := (
3612      {LSstring, Jval, mJval} = basisVec;
3613      Ket[PrettySaundersSLJmJ[basisVec]]
3614    );
3615
3616    LSJMJTemplate =
3617      StringTemplate[
3618      "\!\(\*TemplateBox[{\nRowBox[{\"`LS`\", \",\", \nRowBox[{\"J\", \
3619    \"=\", \"`J`\"}], \",\", \nRowBox[{\"mJ\", \"=\", \"`mJ`\"}]}]},\n\
3620    \"Ket\"]\)"];
3621
3622    BasisVecInLSJMJ::usage = "BasisVecInLSJMJ[basisVec] takes a basis
            vector in the format {{{LSstring, Jval}, mJval}, nucSpin} and
            returns a human-readable symbol for the corresponding LSJMJ term
            in the form |LS, J=..., mJ=...>."
3623    BasisVecInLSJMJ[basisVec_] := (
3624      {LSstring, Jval, mJval} = basisVec;
3625      LSJMJTemplate[<|
3626        "LS" -> LSstring,
3627        "J" -> ToString[Jval, InputForm],
3628        "mJ" -> ToString[mJval, InputForm]|>]
3629    );
3630
3631    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.";
3632    ParseStates[states_, basis_, OptionsPattern[]]:=Module[
3633      {parsedStates},
3634      (
3635        parsedStates = Table[(
3636          {energy, eigenVec}      = state;
3637          maxTermIndex            = Ordering[Abs[eigenVec]][[-1]];
3638          {LSstring, Jval, mJval} = basis[[maxTermIndex]];
3639          LSJsymbol               = Subscript[LSstring, {Jval, mJval}];
3640          LSJMJsymbol             = LSstring <> ToString[Jval,
            InputForm];
3641          {S, L}                  = FindSL[LSstring];
3642          {energy, LSstring, Jval, mJval, S, L, LSJsymbol, LSJMJsymbol}
3643        ),
3644        {state, states}
3645        ];
3646        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"};
  ParseStatesByNumBasisVecs[eigensys_List, basis_List,
    numBasisVecs_Integer, roundTo_Real : 0.01, OptionsPattern[]]:=
    Module[
    {parsedStates, energy, eigenVec,
    probs, amplitudes, ordering,
    chosenIndices, majorComponents,
    majorAmplitudes, majorRep},
    (
      parsedStates = Table[(
        {energy, eigenVec} = state;
        energy             = Chop[energy];
        probs              = Round[Abs[eigenVec^2], roundTo];
        amplitudes         = Round[eigenVec, roundTo];
        ordering           = Ordering[probs];
        chosenIndices      = ordering[[-numBasisVecs ;;]];
        majorComponents    = basis[[chosenIndices]];
        majorThings = If[OptionValue["Coefficients"] == "
    Probabilities",
          (
            probs[[chosenIndices]]
          ),
          (
            amplitudes[[chosenIndices]]
          )
        ];
        majorComponents    = PrettySaundersSLJmJ[#, "Representation"
    -> OptionValue["Representation"]] & /@ majorComponents;
        nonZ               = (# != 0.) & /@ majorThings;
        majorThings        = Pick[majorThings, nonZ];
        majorComponents    = Pick[majorComponents, nonZ];
        If[OptionValue["Coefficients"] == "Probabilities",
          (
            majorThings = majorThings * 100*"%"
          )
        ];
        majorRep           = majorThings . majorComponents;
        {energy, majorRep}
```

```
3687          ),
3688          {state, eigensys}];
3689        Return[parsedStates]
3690      )
3691    ];
3692
3693    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.";
3694    FindThresholdPosition[list_, threshold_] := Module[
3695      {position},
3696      (
3697      position = Position[list, _?(# >= threshold &), 1, 1];
3698      thrPos = If[Length[position] > 0,
3699        position[[1, 1]],
3700        Length[list]];
3701      If[thrPos == 0,
3702        Return[1],
3703        Return[thrPos]]
3704      )
3705    ];
3706
3707    ParseStateByProbabilitySum[{energy_, eigenVec_}, probSum_, roundTo_
        :0.01, maxParts_:20] := Compile[
3708      {{energy, _Real, 0}, {eigenVec, _Complex, 1},
3709       {probSum, _Real, 0},{roundTo, _Real, 0},
3710       {maxParts, _Integer, 0}},
3711        Module[
3712          {numStates, state, amplitudes, probs, ordering,
3713          orderedProbs, truncationIndex, accProb, thresholdIndex,
        chosenIndices, majorComponents,
3714          majorAmplitudes, absMajorAmplitudes, notnullAmplitudes,
        majorRep},
3715          (
3716            numStates     = Length[eigenVec];
3717            (*Round them up*)
3718            amplitudes          = Round[eigenVec, roundTo];
3719            probs               = Round[Abs[eigenVec^2], roundTo];
3720            ordering            = Reverse[Ordering[probs]];
3721            (*Order the probabilities from high to low*)
3722            orderedProbs        = probs[[ordering]];
3723            (*To speed up Accumulate, assume that only as much as
        maxParts will be needed*)
3724            truncationIndex     = Min[maxParts, Length[orderedProbs]];
3725            orderedProbs        = orderedProbs[[;;truncationIndex]];
3726            (*Accumulate the probabilities*)
3727            accProb             = Accumulate[orderedProbs];
3728            (*Find the index of the first element in accProb that is
        greater than probSum*)
3729            thresholdIndex      = Min[Length[accProb],
        FindThresholdPosition[accProb, probSum]];
3730            (*Grab all the indicees up till that one*)
3731            chosenIndices       = ordering[[;; thresholdIndex]];
```

```
3732            (*Select the corresponding elements from the basis*)
3733            majorComponents    = basis[[chosenIndices]];
3734            (*Select the corresponding amplitudes*)
3735            majorAmplitudes    = amplitudes[[chosenIndices]];
3736            (*Take their absolute value*)
3737            absMajorAmplitudes = Abs[majorAmplitudes];
3738            (*Make sure that there are no effectively zero
    contributions*)
3739            notnullAmplitudes  = Flatten[Position[absMajorAmplitudes,
    x_ /; x != 0]];
3740            (* majorComponents    = PrettySaundersSLJmJ
    [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
3741            majorComponents    = PrettySaundersSLJmJ /@ majorComponents
    ;
3742            majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
3743            (*Make them into Kets*)
3744            majorComponents    = Ket /@ majorComponents[[
    notnullAmplitudes]];
3745            (*Multiply and add to build the final Ket*)
3746            majorRep           = majorAmplitudes . majorComponents;
3747            Return[{energy, majorRep}];
3748          )
3749        ],
3750      CompilationTarget -> "C",
3751      RuntimeAttributes -> {Listable},
3752      Parallelization -> True,
3753      RuntimeOptions -> "Speed"
3754    ];
3755
3756    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.";
3757    ParseStatesByProbabilitySum[eigensys_, basis_, probSum_, roundTo_ :
       0.01, maxParts_: 20]:=Module[
3758      {parsedByProb, numStates, state, energy, eigenVec, amplitudes,
      probs, ordering,
3759       orderedProbs, truncationIndex, accProb, thresholdIndex,
      chosenIndices, majorComponents,
3760       majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep
      },
3761       (
3762        numStates    = Length[eigensys];
3763        parsedByProb = Table[(
3764          state              = eigensys[[idx]];
3765          {energy, eigenVec} = state;
3766          (*Round them up*)
3767          amplitudes         = Round[eigenVec, roundTo];
3768          probs              = Round[Abs[eigenVec^2], roundTo];
3769          ordering           = Reverse[Ordering[probs]];
3770          (*Order the probabilities from high to low*)
3771          orderedProbs       = probs[[ordering]];
3772          (*To speed up Accumulate, assume that only as much as
      maxParts will be needed*)
```

```
3773        truncationIndex    = Min[maxParts, Length[orderedProbs]];
3774        orderedProbs       = orderedProbs[[;;truncationIndex]];
3775        (*Accumulate the probabilities*)
3776        accProb            = Accumulate[orderedProbs];
3777        (*Find the index of the first element in accProb that is
    greater than probSum*)
3778        thresholdIndex     = Min[Length[accProb],
    FindThresholdPosition[accProb, probSum]];
3779        (*Grab all the indicees up till that one*)
3780        chosenIndices      = ordering[[;; thresholdIndex]];
3781        (*Select the corresponding elements from the basis*)
3782        majorComponents    = basis[[chosenIndices]];
3783        (*Select the corresponding amplitudes*)
3784        majorAmplitudes    = amplitudes[[chosenIndices]];
3785        (*Take their absolute value*)
3786        absMajorAmplitudes = Abs[majorAmplitudes];
3787        (*Make sure that there are no effectively zero contributions
    *)
3788        notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_
    /; x != 0]];
3789        (* majorComponents    = PrettySaundersSLJmJ
    [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
3790        majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
3791        majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
3792        majorComponents    = majorComponents[[notnullAmplitudes]];
3793        (*Multiply and add to build the final Ket*)
3794        majorRep           = majorAmplitudes . majorComponents;
3795        {energy, majorRep}
3796        ), {idx, numStates}];
3797      Return[parsedByProb];
3798    )
3799  ];
3800
3801  (* ################## Eigensystem analysis ################## *)
3802  (* ########################################################## *)
3803
3804  (* ########################################################## *)
3805  (* ########################## Misc ########################## *)
3806
3807  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.";
3808  SymbToNum[expr_, replacementAssociation_]:= (
3809    includedKeys = Keys[replacementAssociation];
3810    (*If a key is not defined, make its value zero.*)
3811    fullAssociation = Table[(
3812      If[MemberQ[includedKeys, key],
3813        ToExpression[key]->replacementAssociation[key],
3814        ToExpression[key]->0
3815      ]
3816    ),
3817    {key, paramSymbols}];
3818    Return[expr/.fullAssociation];
```

```
);

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.";
SimpleConjugate[expr_] := expr /. Complex[a_, b_] :> a - I b;

ExportMZip::usage="ExportMZip[\"dest.[zip,m]\"] saves a compressed
  version of expr to the given destination.";
ExportMZip[filename_, expr_]:=Module[
  {baseName, exportName, mImportName, zipImportName},
  (
    baseName    = FileBaseName[filename];
    exportName  = StringReplace[filename,".m"->".zip"];
    mImportName = StringReplace[exportName,".zip"->".m"];
    If[FileExistsQ[mImportName],
    (
      PrintTemporary[mImportName<>" exists already, deleting"];
      DeleteFile[mImportName];
      Pause[2];
    )
    ];
    Export[exportName, (baseName<>".m") -> expr];
  )
];

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";
Options[ImportMZip]={"Leave Uncompressed" -> True};
ImportMZip[filename_String, OptionsPattern[]]:=Module[
  {baseName, importKey, zipImportName, mImportName, imported},
  (
    baseName      = FileBaseName[filename];
    (*Function allows for the filename to be .m or .zip*)
    importKey     = baseName <> ".m";
    zipImportName = StringReplace[filename, ".m"->".zip"];
    mImportName   = StringReplace[zipImportName, ".zip"->".m"];
    If[FileExistsQ[mImportName],
    (
      PrintTemporary[".m version exists already, importing that
  instead ..."];
      Return[Import[mImportName]];
    )
    ];
    imported = Import[zipImportName, importKey];
    If[OptionValue["Leave Uncompressed"],
      Export[mImportName, imported]
    ];
```

```
3862        Return[imported]
3863      )
3864    ];
3865
3866    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.";
3867    ReplaceInSparseArray[sparseA_SparseArray, rules_]:=(
3868      SparseArray[Automatic,
3869        sparseA["Dimensions"],
3870        sparseA["Background"] /. rules,
3871        {
3872          1,
3873          {sparseA["RowPointers"], sparseA["ColumnIndices"]},
3874          sparseA["NonzeroValues"] /. rules
3875        }
3876        ]
3877    );
3878
3879    MapToSparseArray::usage = "MapToSparseArray[sparseArray, function]
          takes a sparse array and returns a sparse array after the function
           has been applied to it.";
3880    MapToSparseArray[sparseA_SparseArray, func_]:=Module[
3881      {nonZ, backg, mapped},
3882      (
3883        nonZ   = func /@ sparseA["NonzeroValues"];
3884        backg  = func[sparseA["Background"]];
3885        mapped = SparseArray[Automatic,
3886          sparseA["Dimensions"],
3887          backg,
3888          {
3889            1,
3890            {sparseA["RowPointers"], sparseA["ColumnIndices"]},
3891            nonZ
3892          }
3893        ];
3894        Return[mapped];
3895      )
3896    ];
3897
3898    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.";
3899    Options[ParseTeXLikeSymbol] = {"Form" -> "List"};
3900    ParseTeXLikeSymbol[bigString_, OptionsPattern[]] := Module[
3901      {form, mainSymbol, symbols},
3902      (
3903        form = OptionValue["Form"];
3904        (* parse greek *)
3905        symbols = Table[(
```

```
3906            str = StringReplace[string, {"\\alpha" -> "α",
3907              "\\beta" -> "β",
3908              "\\gamma" -> "γ",
3909              "\\psi" -> "\[Psi]"}];
3910            symbol = Which[
3911              StringContainsQ[str, "_"] && Not[StringContainsQ[str, "^"
    ]],
3912              (
3913                (*yes sub no sup*)
3914                mainSymbol = StringSplit[str, "_"][[1]];
3915                mainSymbol = ToExpression[mainSymbol];
3916
3917                subPart =
3918                  StringCases[str,
3919                    RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
3920                Subscript[mainSymbol, subPart]
3921              ),
3922              Not[StringContainsQ[str, "_"]] && StringContainsQ[str, "^
    "],
3923              (
3924                (*no sub yes sup*)
3925                mainSymbol = StringSplit[str, "^"][[1]];
3926                mainSymbol = ToExpression[mainSymbol];
3927
3928                supPart =
3929                  StringCases[str,
3930                    RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
3931                Superscript[mainSymbol, supPart]
3932              ),
3933              StringContainsQ[str, "_"] && StringContainsQ[str, "^"],
3934              (
3935                (*yes sub yes sup*)
3936                mainSymbol = StringSplit[str, "_"][[1]];
3937                mainSymbol = ToExpression[mainSymbol];
3938                {subPart, supPart} =
3939                  StringCases[str, RegularExpression@"\\{(.*?)\\}" -> "
    $1"];
3940                Subsuperscript[mainSymbol, subPart, supPart]
3941              ),
3942              True,
3943              (
3944                (*no sup or sub*)
3945                str
3946              )
3947              ];
3948            symbol
3949            ),
3950          {string, StringSplit[bigString, " "]}
3951        ];
3952        Which[
3953          form == "Row",
3954          Return[Row[symbols]],
3955          form == "List",
3956          Return[symbols]
3957        ]
```

```
    )
];

(* ########################## 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
  };
EnergyLevelDiagram[states_, OptionsPattern[]]:= (
  energies = First/@states;
  epi = OptionValue["Epilog"];
  explora = If[OptionValue["Explorer"],
    ExploreGraphics,
    Identity
  ];
  explora@ListPlot[Tooltip[{{#, 0}, {#, 1}}, {Quantity
  [#/8065.54429, "eV"], Quantity[#, 1/"Centimeters"]}] &/@ energies,
    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   -> {"\!\(\*FractionBox[\(E\), SuperscriptBox[\(cm
  \), \(-1\)]]\)"},
    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]},
            #1}]} &,
    optAxesRedraw = OptionValue[OptAxesRedraw]},
  DynamicModule[
    {dragging=False, first, second, rx1, rx2, ry1, ry2,
    range = plr},
    {{rx1, rx2}, {ry1, ry2}} = plr;
    Panel@
    EventHandler[
      Dynamic@Graphics[
        If[dragging, {gr, rectangle[first, second]}, gr],
        PlotRange -> Dynamic@range,
        AspectRatio -> ar,
        AxesOrigin -> If[optAxesRedraw,
          Dynamic@Mean[range\[Transpose]], ao],
        Sequence @@ opt],
      {{"MouseDown", 1} :> (
        first = MousePosition["Graphics"]
        ),
      {"MouseDragged", 1} :> (
        dragging = True;
        second = MousePosition["Graphics"]
        ),
      "MouseClicked" :> (
        If[CurrentValue@"MouseClickCount"==2,
          range = plr];
        ),
      {"MouseUp", 1} :> If[dragging,
        dragging = False;

        range = {{rx1, rx2}, {ry1, ry2}} =
          Transpose@{first, second};
        range[[2]] = {0, 1}],
      {"MouseDown", 2} :> (
        first = {sx1, sy1} = MousePosition["Graphics"]
        ),
      {"MouseDragged", 2} :> (
        second = {sx2, sy2} = MousePosition["Graphics"];
        rx1 = rx1 - (sx2 - sx1);
        rx2 = rx2 - (sx2 - sx1);
        ry1 = ry1 - (sy2 - sy1);
        ry2 = ry2 - (sy2 - sy1);
        range = {{rx1, rx2}, {ry1, ry2}};
        range[[2]] = {0, 1};
```

```
4058            )}]]];
4059
4060    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.";
4061    Options[LabeledGrid]={
4062        ItemSize->Automatic,
4063        Alignment->Center,
4064        Frame->All,
4065        "Separator"->",",
4066        "Pivot"->""
4067    };
4068    LabeledGrid[data_,rowHeaders_,columnHeaders_,OptionsPattern[]]:=
          Module[
4069      {gridList=data,rowHeads=rowHeaders,colHeads=columnHeaders},
4070      (
4071        separator=OptionValue["Separator"];
4072        pivot=OptionValue["Pivot"];
4073        gridList=Table[
4074                Tooltip[
4075                  data[[rowIdx,colIdx]],
4076                  DisplayForm[
4077                    RowBox[{rowHeads[[rowIdx]],
4078                            separator,
4079                            colHeads[[colIdx]]}
4080                          ]
4081                          ]
4082                        ],
4083            {rowIdx,Dimensions[data][[1]]},
4084            {colIdx,Dimensions[data][[2]]}];
4085        gridList=Transpose[Prepend[gridList,colHeads]];
4086        rowHeads=Prepend[rowHeads,pivot];
4087        gridList=Prepend[gridList,rowHeads]//Transpose;
4088        Grid[gridList,
4089            Frame->OptionValue[Frame],
4090            Alignment->OptionValue[Alignment],
4091            Frame->OptionValue[Frame],
4092            ItemSize->OptionValue[ItemSize]
4093            ]
4094      )
4095    ];
4096
4097    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.";
4098    Options[HamiltonianForm]={"Separator"->","Pivot"->""}
4099    HamiltonianForm[hamMatrix_, basisLabels_List, OptionsPattern[]]:=(
4100        braLabels=DisplayForm[RowBox[{"\[LeftAngleBracket]",#,"\[
```

```
  RightBracketingBar]"}]]& /@ basisLabels;
4101      ketLabels=DisplayForm[RowBox[{"\[LeftBracketingBar]",#,"\[
  RightAngleBracket]"}]]& /@ basisLabels;
4102      LabeledGrid[hamMatrix,braLabels,ketLabels,"Separator"->
  OptionValue["Separator"],"Pivot"->OptionValue["Pivot"]]
4103      )
4104
4105  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 \"Overlay Values\
  " can be used to specify whether the matrix elements should be
  displayed on top of the matrix plot.";
4106  Options[HamiltonianMatrixPlot] = Join[Options[MatrixPlot], {"Hover"
    -> True, "Overlay Values" -> True}];
4107  HamiltonianMatrixPlot[hamMatrix_, basisLabels_, opts :
  OptionsPattern[]] := (
4108   braLabels = DisplayForm[RowBox[{"\[LeftAngleBracket]", #, "\[
  RightBracketingBar]"}]] & /@ basisLabels;
4109   ketLabels = DisplayForm[Rotate[RowBox[{"\[LeftBracketingBar]", #,
   "\[RightAngleBracket]"}],\[Pi]/2]] & /@ basisLabels;
4110   ketLabelsUpright = DisplayForm[RowBox[{"\[LeftBracketingBar]", #,
   "\[RightAngleBracket]"}]] & /@ basisLabels;
4111   numRows = Length[hamMatrix];
4112   numCols = Length[hamMatrix[[1]]];
4113   epiThings = Which[
4114     And[OptionValue["Hover"], Not[OptionValue["Overlay Values"]]],
4115     Flatten[
4116       Table[
4117         Tooltip[
4118           {
4119             Transparent,
4120             Rectangle[
4121             {j - 1, numRows - i},
4122             {j - 1, numRows - i} + {1, 1}
4123             ]
4124           },
4125         Row[{braLabels[[i]],ketLabelsUpright[[j]],"=",hamMatrix[[i,
   j]]}]]
4126         ],
4127       {i, 1, numRows},
4128       {j, 1, numCols}
4129       ]
4130     ],
4131     And[OptionValue["Hover"], OptionValue["Overlay Values"]],
4132     Flatten[
4133       Table[
4134         Tooltip[
4135           {
4136             Transparent,
4137             Rectangle[
4138             {j - 1, numRows - i},
4139             {j - 1, numRows - i} + {1, 1}
4140             ]
```

```
4141              },
4142            DisplayForm[RowBox[{"\[LeftAngleBracket]", basisLabels[[i
    ]], "\[LeftBracketingBar]", basisLabels[[j]], "\[RightAngleBracket
    ]"}]]
4143            ],
4144          {i, numRows},
4145          {j, numCols}
4146          ]
4147        ],
4148        True,
4149        {}
4150        ];
4151      textOverlay = If[OptionValue["Overlay Values"],
4152        (
4153          Flatten[
4154          Table[
4155            Text[hamMatrix[[i, j]],
4156              {j - 1/2, numRows - i + 1/2}
4157            ],
4158          {i, 1, numRows},
4159          {j, 1, numCols}
4160          ]
4161          ]
4162        ),
4163        {}
4164        ];
4165      epiThings = Join[epiThings, textOverlay];
4166      MatrixPlot[hamMatrix,
4167        FrameTicks -> {
4168          {Transpose[{Range[Length[braLabels]], braLabels}], None},
4169          {None, Transpose[{Range[Length[ketLabels]], ketLabels}]}
4170          },
4171        Evaluate[FilterRules[{opts}, Options[MatrixPlot]]],
4172        Epilog -> epiThings
4173      ]
4174    );
4175
4176    (* ################## Some Plotting Routines ################## *)
4177    (* ############################################################ *)
4178
4179    (* ############################################################ *)
4180    (* ###################### Load Functions ###################### *)
4181
4182    LoadAll::usage="LoadAll[] executes most Load* functions.";
4183    LoadAll[]:=(
4184      LoadTermLabels[];
4185      LoadCFP[];
4186      LoadUk[];
4187      LoadV1k[];
4188      LoadT22[];
4189      LoadSOOandECSOLS[];
4190
4191      LoadElectrostatic[];
4192      LoadSpinOrbit[];
4193      LoadSOOandECSO[];
```

```
4194      LoadSpinSpin[];
4195      LoadThreeBody[];
4196      LoadChenDeltas[];
4197      LoadCarnall[];
4198    );
4199
4200    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.";
4201
4202    LoadTermLabels::usage="LoadTermLabels[] loads into the session the
          labels for the terms in the f^n configurations.";
4203    LoadTermLabels[]:= (
4204      If[ValueQ[fnTermLabels], Return[]];
4205      PrintTemporary["Loading data for state labels in the f^n
          configurations..."];
4206      fnTermsFname = FileNameJoin[{moduleDir, "data", "fnTerms.m"}];
4207
4208      If[!FileExistsQ[fnTermsFname],
4209        (PrintTemporary[">> fnTerms.m not found, generating ..."];
4210          fnTermLabels = ParseTermLabels["Export"->True];
4211        ),
4212        fnTermLabels = Import[fnTermsFname];
4213      ];
4214    );
4215
4216    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}";
4217
4218    LoadCarnall::usage="LoadCarnall[] loads data for trivalent
          lanthanides in LaF3 using the data from Bill Carnall's 1989 paper.
          ";
4219    LoadCarnall[]:=(
4220      If[ValueQ[Carnall], Return[]];
4221      carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
4222      If[!FileExistsQ[carnallFname],
4223        (PrintTemporary[">> Carnall.m not found, generating ..."];
4224          Carnall = ParseCarnall[];
4225        ),
4226        Carnall = Import[carnallFname];
4227      ];
4228    );
4229
4230    LoadChenDeltas::usage="LoadChenDeltas[] loads the differences noted
          by Chen.";
4231    LoadChenDeltas[]:=(
4232      If[ValueQ[chenDeltas], Return[]];
4233      PrintTemporary["Loading the association of discrepancies found by
          Chen ..."];
4234      chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas.m"
```

```
  }];
4235   If[!FileExistsQ[chenDeltasFname],
4236     (PrintTemporary[">> chenDeltas.m not found, generating ..."];
4237       chenDeltas = ParseChenDeltas[];
4238     ),
4239     chenDeltas = Import[chenDeltasFname];
4240   ];
4241 );
4242
4243 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";
4244 Options[ParseChenDeltas] = {"Export" -> True};
4245 ParseChenDeltas[OptionsPattern[]]:=(
4246   chenDeltasRaw = Import[FileNameJoin[{moduleDir, "data", "the-chen
    -deltas-A.csv"}]];
4247   chenDeltasRaw = chenDeltasRaw[[2 ;;]];
4248   chenDeltas = <||>;
4249   chenDeltasA = <||>;
4250   Off[Power::infy];
4251   Do[
4252     ({right, wrong} = {chenDeltasRaw[[row]][[4 ;;]],
4253       chenDeltasRaw[[row + 1]][[4 ;;]]};
4254     key = chenDeltasRaw[[row]][[1 ;; 3]];
4255     repRule = (#[[1]] -> #[[2]]*#[[1]]) & /@
4256       Transpose[{{M0, M2, M4, P2, P4, P6}, right/wrong}];
4257     chenDeltasA[key] = <|"right" -> right, "wrong" -> wrong,
4258       "repRule" -> repRule|>;
4259     chenDeltasA[{key[[1]], key[[3]], key[[2]]}] = <|"right" ->
    right,
4260       "wrong" -> wrong, "repRule" -> repRule|>;
4261     ),
4262     {row, 1, Length[chenDeltasRaw], 2}];
4263   chenDeltas["A"] = chenDeltasA;
4264
4265   chenDeltasRawB = Import[FileNameJoin[{moduleDir, "data", "the-
    chen-deltas-B.csv"}], "Text"];
4266   chenDeltasB = StringSplit[chenDeltasRawB, "\n"];
4267   chenDeltasB = StringSplit[#, ","] & /@ chenDeltasB;
4268   chenDeltasB = {ToExpression[StringTake[#[[1]], {2}]], #[[2]],
    #[[3]]} & /@ chenDeltasB;
4269   chenDeltas["B"] = chenDeltasB;
4270   On[Power::infy];
4271   If[OptionValue["Export"],
4272     (chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas
    .m"}];
4273     Export[chenDeltasFname, chenDeltas];
4274     )
4275     ];
4276   Return[chenDeltas];
4277 );
4278
4279 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).";
4280  Options[ParseCarnall] = {"Export" -> True};
4281  ParseCarnall[OptionsPattern[]] := (
4282    ions          = {"Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho"
      ,"Er","Tm","Yb"};
4283    templates     = StringTemplate/@StringSplit["appendix:`ion`:
      Association appendix:`ion`:Calculated appendix:`ion`:RawTable
      appendix:`ion`:Headings"," "];
4284
4285    (* How many unique eigenvalues, after removing Kramer's
      degeneracy *)
4286    fullSizes     = AssociationThread[ions, {7, 91, 182, 1001, 1001,
      3003, 1716, 3003, 1001, 1001, 182, 91, 7}];
4287    carnall       = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
      "}]][[2]];
4288    carnallErr    = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
      "}]][[3]];
4289
4290    elementNames = carnall[[1]][[2;;]];
4291    carnall       = carnall[[2;;]];
4292    carnallErr    = carnallErr[[2;;]];
4293    carnall       = Transpose[carnall];
4294    carnallErr    = Transpose[carnallErr];
4295    paramNames    = ToExpression/@carnall[[1]][[1;;]];
4296    carnall       = carnall[[2;;]];
4297    carnallErr    = carnallErr[[2;;]];
4298    carnallData   = Table[(
4299                     data          = carnall[[i]];
4300                     data          = (#[[1]]->#[[2]])&/@Select[
      Transpose[{paramNames,data}],#[[2]]!=""&];
4301                     elementNames[[i]]->data
4302                     ),
4303                     {i,1,13}
4304                     ];
4305    carnallData   = Association[carnallData];
4306    carnallNotes = Table[(
4307                     data         = carnallErr[[i]];
4308                     elementName = elementNames[[i]];
4309                     dataFun      = (
4310                         #[[1]] -> If[#[[2]]=="[]",
4311                         "Not allowed to vary in fitting.",
4312                         If[#[[2]]=="[R]",
4313                             "Ratio constrained by: " <> <|"Eu"->"F4/
      F2=0.713; F6/F2=0.512",
4314                                 "Gd"->"F4/F2=0.710]",
4315                                 "Tb"->"F4/F2=0.707"|>[elementName],
4316                             If[#[[2]]=="i",
4317                                 "Interpolated",
4318                                 #[[2]]
4319                             ]
4320                         ]
4321                         ]) &;
4322                     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;
```

```
4373          Carnall[keys[[3]]]  = carnallData;
4374          Carnall[keys[[4]]]  = headers;
4375          ),
4376      {sheetIdx,4,16}
4377      ];
4378
4379      goodions = Select[ions,#!="Pm"&];
4380      expData  = Select[Transpose[Carnall["appendix:"<>#<>":RawTable"
         ]][[1+Position[Carnall["appendix:"<>#<>":Headings"],"Exp (1/cm)"
         ][[1,1]]]],NumberQ]&/@goodions;
4381      Carnall["All Experimental Data"]=AssociationThread[goodions,
         expData];
4382      If[OptionValue["Export"],
4383        (
4384          carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"
         }];
4385          Print["Exporting to "<>carnallFname];
4386          Export[carnallFname, Carnall];
4387        )
4388        ];
4389      Return[Carnall];
4390    );
4391
4392    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";
4393
4394    CFPAssoc::usage = " CFPAssoc is an association where keys are of
       lists of the form {num_electrons, daugherTerm, 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[].";
4395
4396    LoadCFP::usage="LoadCFP[] loads CFP, CFPAssoc, and CFPTable into
       the session.";
4397    LoadCFP[]:=(
4398      If[And[ValueQ[CFP], ValueQ[CFPTable], ValueQ[CFPAssoc]],Return
       []];
4399
4400      PrintTemporary["Loading CFPtable ..."];
4401      CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"}];
4402      If[!FileExistsQ[CFPTablefname],
4403        (PrintTemporary[">> CFPTable.m not found, generating ..."];
4404          CFPTable = GenerateCFPTable["Export"->True];
4405        ),
4406        CFPTable = Import[CFPTablefname];
4407      ];
4408
4409      PrintTemporary["Loading CFPs.m ..."];
4410      CFPfname = FileNameJoin[{moduleDir, "data", "CFPs.m"}];
4411      If[!FileExistsQ[CFPfname],
4412        (PrintTemporary[">> CFPs.m not found, generating ..."];
```

```
4413          CFP = GenerateCFP["Export"->True];
4414        ),
4415        CFP = Import[CFPfname];
4416      ];
4417
4418      PrintTemporary["Loading CFPAssoc.m ..."];
4419      CFPAfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"}];
4420      If[!FileExistsQ[CFPAfname],
4421        (PrintTemporary[">> CFPAssoc.m not found, generating ..."];
4422          CFPAssoc = GenerateCFPAssoc["Export"->True];
4423        ),
4424        CFPAssoc = Import[CFPAfname];
4425      ];
4426    );
4427
4428    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[].";
4429
4430    LoadUk::usage="LoadUk[] loads into session the reduced matrix
       elements for unit tensor operators.";
4431    LoadUk[]:=(
4432      If[ValueQ[ReducedUkTable], Return[]];
4433      PrintTemporary["Loading the association of reduced matrix
       elements for unit tensor operators ..."];
4434      ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
       ReducedUkTable.m"}];
4435      If[!FileExistsQ[ReducedUkTableFname],
4436        (PrintTemporary[">> ReducedUkTable.m not found, generating ..."
       ];
4437          ReducedUkTable = GenerateReducedUkTable[7];
4438        ),
4439        ReducedUkTable = Import[ReducedUkTableFname];
4440      ];
4441    );
4442
4443    ElectrostaticTable::usage = "ElectrostaticTable[{n, SL, SpLp}]
       provides the calculated result of Electrostatic[{n, SL, SpLp}].
       Load using LoadElectrostatic[].";
4444
4445    LoadElectrostatic::usage="LoadElectrostatic[] loads the reduced
       matrix elements for the electrostatic interaction.";
4446    LoadElectrostatic[]:=(
4447      If[ValueQ[ElectrostaticTable], Return[]];
4448      PrintTemporary["Loading the association of matrix elements for
       the electrostatic interaction ..."];
4449      ElectrostaticTablefname = FileNameJoin[{moduleDir, "data", "
       ElectrostaticTable.m"}];
4450      If[!FileExistsQ[ElectrostaticTablefname],
4451        (PrintTemporary[">> ElectrostaticTable.m not found, generating
       ..."];
4452          ElectrostaticTable = GenerateElectrostaticTable[7];
4453        ),
4454        ElectrostaticTable = Import[ElectrostaticTablefname];
```

```
4455      ];
4456    );
4457
4458    LoadV1k::usage="LoadV1k[] loads into session the matrix elements of
          V1k.";
4459    LoadV1k[]:=(
4460      If[ValueQ[ReducedV1kTable], Return[]];
4461      PrintTemporary["Loading the association of matrix elements for
        V1k ..."];
4462      ReducedV1kTableFname = FileNameJoin[{moduleDir, "data", "
        ReducedV1kTable.m"}];
4463      If[!FileExistsQ[ReducedV1kTableFname],
4464        (PrintTemporary[">> ReducedV1kTable.m not found, generating ...
        "];
4465          ReducedV1kTable = GenerateReducedV1kTable[7];
4466        ),
4467        ReducedV1kTable = Import[ReducedV1kTableFname];
4468      ]
4469    );
4470
4471    LoadSpinOrbit::usage="LoadSpinOrbit[] loads into session the matrix
          elements of the spin-orbit interaction.";
4472    LoadSpinOrbit[]:=(
4473      If[ValueQ[SpinOrbitTable], Return[]];
4474      PrintTemporary["Loading the association of matrix elements for
        spin-orbit ..."];
4475      SpinOrbitTableFname = FileNameJoin[{moduleDir, "data", "
        SpinOrbitTable.m"}];
4476      If[!FileExistsQ[SpinOrbitTableFname],
4477        (PrintTemporary[">> SpinOrbitTable.m not found, generating ..."
        ];
4478          SpinOrbitTable = GenerateSpinOrbitTable[7, "Export" -> True];
4479        ),
4480        SpinOrbitTable = Import[SpinOrbitTableFname];
4481      ]
4482    );
4483
4484    LoadSOOandECSOLS::usage="LoadSOOandECSOLS[] loads into session the
        LS reduced matrix elements of the SOO-ECSO interaction.";
4485    LoadSOOandECSOLS[]:=(
4486      If[ValueQ[SOOandECSOLSTable], Return[]];
4487      PrintTemporary["Loading the association of LS reduced matrix
        elements for SOO-ECSO ..."];
4488      SOOandECSOLSTableFname = FileNameJoin[{moduleDir, "data", "
        ReducedSOOandECSOLSTable.m"}];
4489      If[!FileExistsQ[SOOandECSOLSTableFname],
4490        (PrintTemporary[">> ReducedSOOandECSOLSTable.m not found,
        generating ..."];
4491          SOOandECSOLSTable = GenerateSOOandECSOLSTable[7];
4492        ),
4493        SOOandECSOLSTable = Import[SOOandECSOLSTableFname];
4494      ];
4495    );
4496
4497    LoadSOOandECSO::usage="LoadSOOandECSO[] loads into session the LSJ
```

```
  reduced matrix elements of spin-other-orbit and electrostatically-
  correlated-spin-orbit.";
LoadSOOandECSO[]:=(
  If[ValueQ[SOOandECSOTableFname], Return[]];
  PrintTemporary["Loading the association of matrix elements for
  spin-other-orbit and electrostatically-correlated-spin-orbit ..."
  ];
  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
   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
    [14, "Export" -> True];
      ),
      ThreeBodyTable = Import[ThreeBodyFname];
      ThreeBodyTables = Import[ThreeBodiesFname];
    ];
  );

  (* ###################### Load Functions ###################### *)
  (* ########################################################### *)

End[];

LoadTermLabels[];
LoadCFP[];

EndPackage[];
```

12.2 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.
*)

Get["qlanth.m"]
Get["qonstants.m"];
Get["misc.m"];
LoadCarnall[];

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. The notebook is created with the title \"
    Solver Progress\". The notebook is created with the option
    WindowSelected->True. The notebook is created with the option
    TextAlignment->Center. The notebook is created with the option
    WindowTitle->\"Solver Progress\".";
Options[ProgressNotebook] = {"Basic" -> True};
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}
        ]
       }
      ),
     WindowSize -> {600, 1000},
     WindowSelected -> True,
     TextAlignment -> Center,
     WindowTitle -> "Solver Progress"
     ],
   True,
   CreateDocument[(
      {
        "",
        Dynamic[Framed[progressMessage]],
        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}],
       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 *)
181     coarseStates  = {#[[1]],#[[-1]]}& /@ coarseStates;
182     missingLevels = Length[coarseStates]-Length[expData];
183
184     (* The energies are in the first element of the tuples. *)
185     energyDiffFun = ( Abs[#1[[1]]-#2[[1]]] ) &;
186
187     (* matching disregarding labels to get overall scale for scaling
        differences in labels *)
188     energyFlow    = FlowMatching[coarseStates,
189                       expData,
190                       \"notMatched\" -> missingLevels,
191                       \"CostFun\"    -> energyDiffFun
192                       ];
193     energyPairs   = {#[[1]][[1]], #[[2]][[1]]}&/@energyFlow[[1]];
194     energyRms     = Sqrt[Total[(Abs[#[[2]]-#[[1]]])^2 & /@
        energyPairs]/Length[energyPairs]];
195
196     (* matching using both labels and energies  *)
197     energyAndLabelFun  = With[{del=energyRms},
198                 (Abs[#1[[1]]-#2[[1]]] +
199                 If[#1[[2]]==#2[[2]],
200                   0.,
201                   del])&];
202
203     (* energyAndLabelFun  = With[{del=energyRms},
204                 (Abs[#1[[1]]-#2[[1]]] +
205                 del*EditDistance[#1[[2]],#2[[2]]])&]; *)
206     energyAndLabelFun  = ( Abs[#1[[1]] - #2[[1]]] + EditDistance
        [#1[[2]],#2[[2]]] )&;
207     energyAndLabelFlow = FlowMatching[coarseStates,
208                         expData,
209                         \"notMatched\" -> missingLevels,
210                         \"CostFun\"    -> energyAndLabelFun
211                         ];
212     totalAvgCost       = Total[energyAndLabelFun@@# & /@
        energyAndLabelFlow[[1]]]/Length[energyAndLabelFlow[[1]]];
213     Return[totalAvgCost];
214   )
215 ]"];
216
217 truncatedEnergyCostTemplate = StringTemplate["
218 TheTruncatedAndSignedPath[`varsWithNumericQ`] :=
219 (
220   (* Calculate the truncated Hamiltonian *)
221   numericalFreeIonHam = compileIntermediateTruncatedHam[`
        varsMixedWithFixedVals`];
222
223   (* Diagonalize it *)
224   {truncatedEigenvalues, truncatedEigenVectors} = Eigensystem[
        numericalFreeIonHam];
225
226   (* Using the truncated eigenvectors push them up to the full state
        space *)
227   pulledTruncatedEigenVectors = truncatedEigenVectors.Transpose[
```

```
    truncatedIntermediateBasis];
  states = Transpose[{truncatedEigenvalues,
    pulledTruncatedEigenVectors}];
  states = SortBy[states, First];
  states = ShiftedLevels[states];

  (* Coarsen the resulting eigenstates *)
  coarseStates = ParseStates[states, basis];

  (* Grab the parts that are needed for fitting *)
  coarseStates = {#[[1]], #[[-1]]} & /@ coarseStates;

  (* This cost function takes into account both labels and energies a
     random factor is added for the sake of stability of the solver*)
  energyAndLabelFun = (
        (
          Abs[#1[[1]] - #2[[1]]] +
          EditDistance[#1[[2]], #2[[2]]]
        ) *
        (1 + RandomReal[{-10^-6, 10^-6}])) &;

  (* This one only takes into account the energies *)
  energyFun = (Abs[#1[[1]] - #2[[1]]]*(1 + RandomReal[{0, 10^-6}]) &)
    ;

  (* Choose which cost function to use *)
  costFun = energyAndLabelFun;

  (* Not all states are to be matched to the experimental data *)
  missingLevels = Length[coarseStates] - Length[expData];

  (* If there are more experimental data than calculated ones, don't
    leave any state unmatched to those*)
  missingLevels = If[missingLevels < 0, 0, missingLevels];

  (* Apply the Hungarian algorithm to match the two sets of data *)
  energyAndLabelFlow = FlowMatching[coarseStates,
    expData,
    \"notMatched\" -> missingLevels,
    \"CostFun\" -> costFun];
  totalCosts   = (costFun @@ #)& /@ energyAndLabelFlow[[1]];
  totalAvgCost = Total[totalCosts] / 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]]
  )

LogSol::usage = "LogSol[expr, solHistory, 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_] := (
   fname = prefix <> "-sols-" <> CreateUUID[] <> ".m";
   fname = FileNameJoin[{".", "log", fname}];
   Print["Saving solution to: ", fname];
   Export[fname, theSolution];
   Return[fname];
   );


FitToHam::usage = "FitToHam[numE, expData, fitToSymbols, simplifier,
    OptionsPattern[]] fits the model Hamiltonian to the experimental
    data for the trivalent lanthanide ion with number numE. The
    experimental data is given in the form of a list of tuples. The
    first element of the tuple is the energy and the second element is
     the label. The function saves the results to a file, with the
    string filePrefix prepended to it, by default this is an empty
    string, in which case the filePrefix is modified to be the name of
     the lanthanide.
The fitToSymbols is a list of the symbols to be fit. The simplifier
    is a list of rules that simplify the Hamiltonian.
The options and their defaults are:
\"PrintFun\"->PrintTemporary,
\"FilePrefix\"->\"\",
\"SlackChannel\"->None,
\"MaxHistory\"->100,
\"MaxIters\"->100,
\"NumCycles\"->10,
\"ProgressWindow\"->True
The PrintFun option is the function used to print progress messages.
The FilePrefix option is the prefix to use for the file name, by
    default this is the symbol for the lanthanide.
The SlackChannel option is the channel to post progress messages to.
The MaxHistory option is the maximum number of iterations to keep in
    the history.
The MaxIters option is the maximum number of iterations for the
    solver.
The NumCycles option is the number of cycles to run the solver for.
The function returns a list of solutions. The solutions are the
    results of the NMinimize function. The solutions are a list of
    tuples. The first element of the tuple is the RMS error and the
    second element is the parameter values
The function also saves the solutions to a file. The file is named
    with a prefix and a UUID. The file is saved in the current
    directory. The file is saved in the format of a .m file.";
Options[FitToHam] = {
      "PrintFun"-> PrintTemporary,
      "FilePrefix" -> "",
      "SlackChannel" -> None,
      "MaxHistory" -> 100,
      "ProgressWindow"-> True,
      "MaxIters" -> 100,
      "NumCycles" -> 10};
FitToHam[numE_Integer, expData_List, fitToSymbols_List,
```

```
    simplifier_List, OptionsPattern[]] :=
(
    PrintFun         = OptionValue["PrintFun"];
    fitToVars        = ToExpression[ToString[#] <> "v"] & /@
    fitToSymbols;
    stringfitToVars  = ToString /@ fitToVars;
    slackChan        = OptionValue["SlackChannel"];
    maxHistory       = OptionValue["MaxHistory"];
    maxIters         = OptionValue["MaxIters"];
    numCycles        = OptionValue["NumCycles"];
    ln               = theLanthanides[[numE]];
    logFilePrefix    = If[OptionValue["FilePrefix"] == "",
                          ToString[theLanthanides[[numE]]],
                          OptionValue["FilePrefix"]];
    PrintFun["Assembling the Hamiltonian for f^", numE, " ..."];
    ham = HamMatrixAssembly[numE];
    PrintFun["Simplifying the symbolic expression for the Hamiltonian
     in terms of the given simplifier..."];
    ham = ReplaceInSparseArray[ham, simplifier];
    PrintFun["Determining the variables to be fit for ..."];
    (* as they remain after simplifying *)
    fitVars = Variables[Normal[ham]];
    (* append v to symbols *)
    varVars = ToExpression[ToString[#] <> "v"] & /@ fitVars;

    PrintFun[
    "Compiling a function for efficient evaluation of the Hamiltonian
     matrix ..."];
    compHam = Compile[Evaluate[fitVars], Evaluate[N[Normal[ham]]]];

    PrintFun[
    "Defining the cost function according to given energies and state
     labels ..."];

    varPatterns = StringJoin[{ToString[#], "_?NumericQ"}] & /@
    fitVars;
    varPatterns = Riffle[varPatterns, ", "];
    varPatterns = StringJoin[varPatterns];
    vars = ToString[#] & /@ fitVars;
    vars = Riffle[vars, ", "];
    vars = StringJoin[vars];

    basis = BasisLSJMJ[numE];

    (* define the cost functions given the problem variables *)
    energyCostFunString =
    energyCostFunTemplate[<|
      "varPatterns" -> varPatterns,
      "vars" -> vars|>];
    ToExpression[energyCostFunString];
    energyAndLabelCostFunString = energyAndLabelCostFunTemplate[<|"
    varPatterns" -> varPatterns, "vars" -> vars|>];
    ToExpression[energyAndLabelCostFunString];

    PrintFun["getting starting values from LaF3..."];
```

```
    lnParams = LoadParameters[ln];
    bills = Table[lnParams[ToExpression[StringTake[ToString[varvar],
   {1, -2}]]], {varvar, varVars}];

    (* define the function arguments with the frozen args in place *)
    activeArgs = Table[
      If[MemberQ[fitToVars, varvar],
      varvar,
      lnParams[ToExpression[StringTake[ToString[varvar], {1, -2}]]]],
    {varvar, varVars}
    ];
    activeArgs = StringJoin[Riffle[ToString /@ activeArgs, ", "]];
    (* the constraints, very important *)
    constraints = N[Constrainer[fitToVars, ln]];
    complementaryArgs = Table[
      If[MemberQ[fitToVars, varvar],
      varvar,
      lnParams[ToExpression[StringTake[ToString[varvar], {1, -2}]]]],
    {varvar, varVars}
    ];

    fromBill = {B02v -> B02, B04v -> B04, B06v -> B06, B22v -> B22,
    B24v -> B24, B26v -> B26, B44v -> B44, B46v -> B46, B66v -> B66,
    M0v -> M0, P2v -> P2} /. lnParams;

    If[Not[ValueQ[noteboo]] && OptionValue["ProgressWindow"],
    noteboo = ProgressNotebook["Basic" -> False];
    ];

    threadHeaderTemplate = StringTemplate[
    "(`idx`/`reps`) Fitting data for `ln` using `freeVars`."
    ];
    solutions = {};
    Do[
    (
      (* Remove the downvalues of the cost function *)
      (* DownValues[TheRightSignedPath] = {DownValues[
   TheRightSignedPath][[-1]]}; *)
      (* start history anew *)
      rmsHistory = {};
      paramSols  = {};
      startTime  = Now;
      threadMessage = threadHeaderTemplate[
          <|"reps" -> numCycles,
          "idx" -> rep,
          "ln" -> ln,
          "freeVars" -> ToString[fitToVars]|>];
      If[slackChan =!= None,
        threadTS = PostMessageToSlack[threadMessage, slackChan]["ts"]
      ];
      solverTemplateNMini = StringTemplate["
        numIter = 0;
        sol = NMinimize[
          Evaluate[
            Join[{TheRightSignedPath[`activeArgs`]},
```

```
            constraints
          ]
        ],
        fitToVars,
        MaxIterations->`maxIterations`,
        Method->`Method`,
        `Monitor`:>(
          currentErr = TheRightSignedPath[`activeArgs`];
          numIter   += 1;
          rmsHistory = AddToList[rmsHistory, currentErr, maxHistory, False];
          paramSols  = AddToList[paramSols, fitToVars, maxHistory, False];
        )
      ]
    "];
    solverCode = solverTemplateNMini[<|
      "maxIterations" -> maxIters,
      "Method" -> "{\"DifferentialEvolution\",
          \"PostProcess\"   -> False,
          \"ScalingFactor\" -> 0.9,
          \"RandomSeed\"    -> RandomInteger[{0,1000000}],
          \"SearchPoints\"  -> 10}",
      "Monitor" -> "StepMonitor",
      "activeArgs" -> activeArgs|>];
    ToExpression[solverCode];
    timeTaken = QuantityMagnitude[Now - startTime, "Seconds"];
    Print["Took " <> ToString[timeTaken] <> "s"];
    Print[sol];
    {bestError, bestParams} = sol;
    resultMessage = "sigma=" <> ToString[Round[bestError, 0.1]];
    logFname = LogSol[sol, logFilePrefix];
    If[slackChan =!= None,
    (
      PostMessageToSlack[resultMessage,   slackChan, "threadTS" -> threadTS];
      PostFileToSlack[logFname, logFname, slackChan, "threadTS" -> threadTS];
    )
    ];

    vsBill = TableForm[
    Transpose[{
      First /@ fromBill,
      Last /@ fromBill,
      Round[Last /@ bestParams, 1.]}],
    TableHeadings -> {None, {"Param", "Bill Bkq", "ql Bkq"}}
    ];
    If[slackChan =!= None,
      PostPdfToSlack[logFname, vsBill, slackChan, "threadTS" -> threadTS]
    ];

    (* analysis code *)

```

```
512       finalHam = compHam @@ (complementaryArgs /. bestParams);
513       {eigenEnergies, eigenVecs} = Eigensystem[finalHam];
514       ordering = Ordering[eigenEnergies];
515       eigenEnergies = eigenEnergies - Min[eigenEnergies];
516       states = Transpose[{eigenEnergies, eigenVecs}];
517       states = states[[ordering]];
518       coarseStates = ParseStates[states, basis];
519
520       (* The eigenvectors need to be simplified in order to compare
      labels to labels *)
521       coarseStates = {#[[1]], #[[-1]]} & /@ coarseStates;
522       missingLevels = Length[coarseStates] - Length[expData];
523       (* The energies are in the first element of the tuples. *)
524       energyDiffFun = (Abs[#1[[1]] - #2[[1]]]) &;
525       (* matching disregarding labels to get overall scale for
      scaling differences in labels *)
526       energyFlow = FlowMatching[coarseStates,
527               expData,
528               "notMatched" -> missingLevels,
529               "CostFun" -> energyDiffFun];
530       energyPairs = {#[[1]][[1]], #[[2]][[1]]} & /@ energyFlow[[1]];
531       energyRms = Sqrt[Total[(Abs[#[[2]] - #[[1]]])^2 & /@
      energyPairs] / Length[energyPairs]];
532       (* matching using both labels and energies *)
533       energyAndLabelFun = (Abs[#1[[1]] - #2[[1]]] + EditDistance
      [#1[[2]],#2[[2]]]) &;
534       energyAndLabelFlow = FlowMatching[coarseStates,
535             expData,
536             "notMatched" -> (Length[coarseStates] - Length[expData]),
537             "CostFun" -> energyAndLabelFun];
538       totalAvgCost = Total[energyAndLabelFun @@ # & /@
      energyAndLabelFlow[[1]]] / Length[energyAndLabelFlow[[1]]];
539
540       compa = (Flatten /@ energyAndLabelFlow[[1]]);
541       compa = Join[
542           #,
543           {
544             #[[2]] == #[[4]],
545             If[NumberQ[#[[1]]],
546               Round[#[[1]] - #[[3]],1],
547               ""
548             ],
549           #[[5]] - #[[3]],
550           Which[
551             Round[Abs[#[[1]] - #[[3]]]] < Round[Abs[#[[5]] -
      #[[3]]]],
552               "Better",
553             Round[Abs[#[[1]] - #[[3]]]] == Round[Abs[#[[5]] -
      #[[3]]]],
554               "Equal",
555             True,
556               "Worse"
557           ]
558           }
559       ] & /@ compa;
```

```
560       atable = TableForm[compa,
561         TableHeadings -> {None,
562           {"ql", "ql", "Bill (exp)", "Bill (exp)",
563           "Bill (calc)", "labels=", "ql - exp", "bill - exp"}}
564       ];
565       atable = Framed[atable, FrameMargins -> 20];
566       upsAndDowns = {
567         {"Better", Length[Select[compa, #[[-1]] == "Better" &]]},
568         {"Equal",  Length[Select[compa, #[[-1]] == "Equal" &]]},
569         {"Worse",  Length[Select[compa, #[[-1]] == "Worse" &]]}
570       };
571       upsAndDowns = TableForm[upsAndDowns];
572       If[slackChan =!= None,
573         PostPdfToSlack["table", atable, slackChan, "threadTS" ->
    threadTS];
574         ];
575       solutions = Append[solutions, sol];
576     ),
577     {rep, 1, numCycles}
578     ];
579   )
580
581 TruncationFit::usage="TruncaationFit[numE, expData, numReps,
    activeVars, startingValues, Options] fits the given expData in an
    f^numE configuration, generating numReps different solutions, and
    varying the symbols in activeVars. The list startingValues is a
    list with all of the parameters needed to define the Hamiltonian (
    including values for activeVars, which will be disregarded but are
     required as position placeholders). The function returns a list
    of solutions. The solutions are the results of the NMinimize
    function using the Differential Evolution method. The solutions
    are a list of tuples. The first element of the tuple is the RMS
    error and the second element is a list of replacement rules for
    the fitted parameters. Once each NMinimize is done, the function
    saves the solutions to a file. The file is named with a prefix and
     a UUID. The file is saved in the log sub-directory as a .m file.
    The solver is always constrained by the relevant subsets of
    constraints for the parameters as provided by the Constrainer
    function. By default the Differential Evolution method starts with
     a generation of points within the given constraints, however it
    is also possible here to have a different region from which the
    initial points are chosen with the option \"StartingForVars\".
582
583 The following options can be used:
584   \"SignatureCheck\" : if True then then the function ends
      prematurely, printing 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
       startingValues.
585   \"FilePrefix\" : the prefix to use for the file name, by default
      this is the symbol for the lanthanide.
586   \"AccuracyGoal\": sets the accuracy goal for NMinimize, the default
       is 3.
587   \"MaxHistory\" : determines how long the logs for the solver can be
```

```
    .
  \"MaxIterations\": determines the maximum number of iterations used
    by NMinimize.

  \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
    3.
  \"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 truncated to this value.
  \"PrintFun\": the function used to print progress messages, the
    default is PrintTemporary.
  \"SlackChannel\": name of the Slack channel to which to dump
    progress messaages, the default is None which disables this option
     entirely.
  \"ProgressView\": whether or not a progress window will be opened
    to show the progress of the solver, the default is True.

  \"ReturnHashFileNameAndExit\": if True then the function returns
    the name of the file with the solutions and exits, the default is
    False.
  \"StartingForVars\": if different from {} then it has to be a list
    with two elements. The first element being a number that
    determines the fraction half-width of the interval used for
    choosing the initial generation of points. The second element
    being a list with as many elements as activeVars corresponding to
    the midpoints from which the intial generation points are chosen.
    The default is {}.
  \"DE:CrossProbability\": the cross probability used by the
    Differential Evolution method, the default is 0.5.
  \"DE:ScalingFactor\": the scaling factor used by the Differential
    Evolution method, the default is 0.6.
  \"DE:SearchPoints\": the number of search points used by the
    Differential Evolution method, the default is Automatic.

  \"MagneticSimplifier\": a list of replacement rules to simplify the
     Marvin and pesudo-magnetic paramters.
  \"MagFieldSimplifier\": a list of replacement rules to specify a
    magnetic field (in T), if set to {}, then {Bx, By, Bz} can also
    then be used as variables to be fitted for.
  \"SymmetrySimplifier\": a list of replacements rules to simplify
    the crystal field.
  \"OtherSimplifier\": an additiona list of replacement rules that
    are applied to the Hamiltonian before computing with it.
  \"ThreeBodySimplifier\": the default is an Association that simply
    states which three body parameters Tk are zero in different
    configurations, if a list of replacement rules is used then that
    is used instead for the given problem.

  \"FreeIonSymbols\": a list with the symbols to be included in the
    intermediate coupling basis.
  \"AppendToLogFile\": an association appended to the log file under
    the key \"Appendix\".
  ";
Options[TruncationFit]={
  "MaxHistory"       -> 200,
```

```
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "",
  "AccuracyGoal"     -> 3,
  "TruncationEnergy" -> Automatic,
  "PrintFun"         -> PrintTemporary,
  "SlackChannel"     -> None,
  "ProgressView"     -> True,
  "SignatureCheck"   -> False,
  "AppendToLogFile"  -> <||>,
  "StartingForVars"  -> {},
  "ReturnHashFileNameAndExit" -> False,
  "DE:CrossProbability" -> 0.5,
  "DE:ScalingFactor"    -> 0.6,
  "DE:SearchPoints"     -> 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,T11->0,T12->0,T14->0,T15->0,
    T16->0,T18->0,T17->0,T19->0,T2p->0
    },
"ThreeBodySimplifier" -> <|
    1  -> {
      T2->0,T3->0,T4->0,T6->0,T7->0,T8->0,T11p->0,T11->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,T11->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,T11->0,T12->0,T14->0,T15
    ->0,T16->0,T18->0,T17->0,T19->0,T2p->0
      },
    13->{
```

```
664       T2->0,T3->0,T4->0,T6->0,T7->0,T8->0,T11p->0,T11->0,T12->0,T14
      ->0,T15->0,T16->0,T18->0,T17->0,T19->0,T2p->0
665       }
666     |>,
667 "FreeIonSymbols" -> {F0, F2, F4, F6, M0, P2, α, β, γ, ζ, T2, T3, T4,
      T6, T7, T8}
668 };
669 TruncationFit[numE_Integer, expData0_List, numReps_Integer,
      activeVars_List, startingValues_List, OptionsPattern[]]:=(
670   ln               = theLanthanides[[numE]];
671   expData          = expData0;
672   PrintFun         = OptionValue["PrintFun"];
673   truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic,
674     Max[First/@expData],
675     OptionValue["TruncationEnergy"]
676   ];
677   oddsAndEnds    = <||>;
678   expData        = Select[expData, #[[1]] <= truncationEnergy &];
679   maxIterations  = OptionValue["MaxIterations"];
680   maxHistory     = OptionValue["MaxHistory"];
681   slackChan      = OptionValue["SlackChannel"];
682   accuracyGoal   = OptionValue["AccuracyGoal"];
683   logFilePrefix  = If[OptionValue["FilePrefix"] == "",
684                       ToString[theLanthanides[[numE]]],
685                       OptionValue["FilePrefix"]];
686
687   usingInitialRange  = Not[OptionValue["StartingForVars"] === {}];
688   If[usingInitialRange,
689     (
690       PrintFun["Using the solver for initial values in range ..."];
691       {fractionalWidth, startVarValues} = OptionValue["
      StartingForVars"];
692     )
693   ];
694
695   magneticSimplifier   = OptionValue["MagneticSimplifier"];
696   magFieldSimplifier   = OptionValue["MagFieldSimplifier"];
697   symmetrySimplifier   = OptionValue["SymmetrySimplifier"];
698   otherSimplifier      = OptionValue["OtherSimplifier"];
699   threeBodySimplifier  = If[Head[OptionValue["ThreeBodySimplifier"]]
      == Association,
700     OptionValue["ThreeBodySimplifier"][numE],
701     OptionValue["ThreeBodySimplifier"]
702   ];
703   simplifier           = Join[magneticSimplifier,
704                           magFieldSimplifier,
705                           symmetrySimplifier,
706                           threeBodySimplifier,
707                           otherSimplifier];
708   freeIonSymbols = OptionValue["FreeIonSymbols"];
709   runningInteractive = (Head[$ParentLink] === LinkObject);
710
711   oddsAndEnds["simplifier"] = simplifier;
712   oddsAndEnds["freeIonSymbols"] = freeIonSymbols;
713   oddsAndEnds["truncationEnergy"] = truncationEnergy;
```

```
714   oddsAndEnds["numE"] = numE;
715   oddsAndEnds["expData"] = expData;
716   oddsAndEnds["numReps"] = numReps;
717   oddsAndEnds["activeVars"] = activeVars;
718   oddsAndEnds["startingValues"] = startingValues;
719   oddsAndEnds["maxIterations"] = maxIterations;
720   oddsAndEnds["PrintFun"] = PrintFun;
721   oddsAndEnds["ln"] = ln;
722   oddsAndEnds["numE"] = numE;
723   oddsAndEnds["accuracyGoal"] = accuracyGoal;
724   oddsAndEnds["Appendix"] = OptionValue["AppendToLogFile"];
725
726   hamDim         = Binomial[14, numE];
727   modelSymbols   = Sort[Select[paramSymbols, Not[MemberQ[Join[
        racahSymbols, chenSymbols,{t2Switch, \[Epsilon],gs}],#]]&]];
728   (* Remove the symbols that will be removed by the simplifier, no
        symbol should remain here that is not in the symbolic hamiltonian
        *)
729   reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
        simplifier],#]]&];
730   If[OptionValue["SignatureCheck"],
731     (
732     Print["Given the model parameters and the simplifying assumptions
        , the resultant model parameters are:"];
733     Print[{reducedModelSymbols}];
734     Print["The ordering in these needs to be respected in the
        startValues parameter ..."];
735     Print["Exiting ..."];
736     Return[""];
737     )
738   ];
739
740   (*calculate the basis*)
741   basis          = BasisLSJMJ[numE];
742   (* grab the Hamiltonian preserving its block structure *)
743   PrintFun["Assembling the Hamiltonian for f^",numE," keeping the
        block structure ..."];
744   ham            = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
745   (* apply the simplifier *)
746   PrintFun["Simplifying using the given aggregate set of
        simplification rules ..."];
747   ham            = Map[ReplaceInSparseArray[#, simplifier]&, ham, {2}];
748
749   (* Get the reference parameters from LaF3 *)
750   PrintFun["Getting reference parameters for ",ln," using LaF3 ..."];
751   lnParams = LoadParameters[ln];
752   freeBies = Prepend[Values[(#->(#/.lnParams))&/@freeIonSymbols],numE
        ];
753   (* a more explicit alias *)
754   allVars            = reducedModelSymbols;
755
756   oddsAndEnds["allVars"]  = allVars;
757   oddsAndEnds["freeBies"] = freeBies;
758
759   (* reload compiled version if found *)
```

```
760   varHash                   = Hash[{numE, allVars, freeBies,
        truncationEnergy}];
761   compileIntermediateFname = "compileIntermediateTruncatedHam-"<>
        ToString[varHash]<>".mx";
762   truncadedFname            = "TheTruncatedAndSignedPath-"<>ToString[
        varHash]<>".mx";
763   If[OptionValue["ReturnHashFileNameAndExit"],
764     (
765       Print[varHash];
766       Return[truncadedFname];
767     )
768   ];
769   If[FileExistsQ[compileIntermediateFname],
770     PrintFun["This ion and free-ion params have been compiled before
        (as determined by {numE, allVars, freeBies, truncationEnergy}).
        Loading the previously saved function and intermediate coupling
        basis ..."];
771     {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
        Import[compileIntermediateFname];,
772   (
773     PrintFun["Zeroing out every symbol in the Hamiltonian that is not
         a free-ion parameter ..."];
774     (* Get the free ion symbols *)
775     freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
        freeIonSymbols];
776     (* Take the diagonal blocks for the intermediate analysis *)
777     PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."];
778     diagonalBlocks       = Diagonal[ham];
779     (* simplify them to only keep the free ion symbols *)
780     PrintFun["Simplifying the diagonal blocks to only keep the free
        ion symbols ..."];
781     diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
        ]&/@diagonalBlocks;
782     (* these include the MJ quantum numbers, remove that *)
783     PrintFun["Contracting the basis vectors by removing the MJ
        quantum numbers from the diagonal blocks ..."];
784     diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
785
786     argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
        Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
787     argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
        ToString[#]<>"v") & /@ freeIonSymbols,", "]];
788     PrintFun["argsOfTheIntermediateEigensystems = ",
        argsOfTheIntermediateEigensystems];
789     PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
        argsForEvalInsideOfTheIntermediateSystems];
790     PrintFun["If the following fails, make sure to modify the
        arguments of TheIntermediateEigensystems to match the ones above
        ..."];
791
792     (* 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 *)
793     (* Compile one function for each of the blocks *)
794     PrintFun["Compiling functions for the diagonal blocks of the
```

```
    Hamiltonian ..."];
795  compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N[
    Normal[#]]]]&/@diagonalScalarBlocks;
796  (* Use that to create a function that will calculate the free-ion
     eigensystem *)
797  TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, M0v_,
     P2v_, αv_, βv_, γv_, ζv_, T2v_, T3v_, T4v_, T6v_, T7v_, T8v_] :=
    (
798    theNumericBlocks = (#[F0v, F2v, F4v, F6v, M0v, P2v, αv, βv, γv,
     ζv, T2v, T3v, T4v, T6v, T7v, T8v]&)/@compiledDiagonal;
799    theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
800    Js      = AllowedJ[numEv];
801    basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
802    (* Having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
803    elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2Js
    [[#2[[1]]]]+1]&, theIntermediateEigensystems];
804    pivot = If[EvenQ[numEv],0,-1/2];
805    LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/@Select[
    BasisLSJMJ[numEv],#[[-1]]== pivot &];
806    (* Calculate the multiplet assignments that the intermediate
    basis eigenvectors have *)
807    multipletAssignments = Table[
808      (
809        J          = Js[[idx]];
810        eigenVecs = theIntermediateEigensystems[[idx]][[2]];
811        majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
812        majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
813        (* All of the degenerate eigenvectors belong to the same
    multiplet*)
814        elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments,2J+1];
815        elevatedMultipletAssignments
816      ),
817    {idx, 1, Length[Js]}
818    ];
819    (* Put together the multiplet assignments and the energies *)
820    freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
821    freeIenergiesAndMultiplets = Flatten[freeIenergiesAndMultiplets
    , 1];
822    (* Calculate the change of basis matrix using the intermediate
    coupling eigenvectors *)
823    basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
824    basisChanger = SparseArray[basisChanger];
825    Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
826  );
827
828  PrintFun["Calculating the intermediate eigensystems for ",ln,"
    using free-ion params from LaF3 ..."];
```

```
829     (* Calculate intermediate coupling basis using the free-ion
        params for LaF3 *)
830     {theIntermediateEigensystems, multipletAssignments,
        elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
        basisChanger} = TheIntermediateEigensystems@@freeBies;
831
832     (* Use that intermediate coupling basis to compile a function for
         the full Hamiltonian *)
833     allFreeEnergies = Flatten[First/@elevatedIntermediateEigensystems
        ];
834     (* Important that the intermediate coupling basis have attached
        energies, which make possible the truncation *)
835     ordering = Ordering[allFreeEnergies];
836     (* Sort the free ion energies and determine which indices should
        be included in the truncation *)
837     allFreeEnergiesSorted          = Sort[allFreeEnergies];
838     {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
839     (* Determine the index at which the energy is equal or larger
        than the truncation energy *)
840     sortedTruncationIndex = Which[
841       truncationEnergy > (maxFreeEnergy-minFreeEnergy),
842       hamDim,
843       True,
844       FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted],
        x_/;x>truncationEnergy,{0},1][[1]]
845     ];
846     (* The actual energy at which the truncation is made *)
847     roundedTruncationEnergy = allFreeEnergiesSorted[[
        sortedTruncationIndex]];
848
849     (* The indices that enact the truncation *)
850     truncationIndices = ordering[[;;sortedTruncationIndex]];
851
852     (* Using the ham (with all the symbols) change the basis to the
        computed one *)
853     PrintFun["Changing the basis of the Hamiltonian to the
        intermediate coupling basis ..."];
854     intermediateHam            = Transpose[basisChanger].ArrayFlatten
        [ham].basisChanger;
855     (* Using the truncation indices truncate that one *)
856     PrintFun["Truncating the Hamiltonian ..."];
857     truncatedIntermediateHam   = intermediateHam[[truncationIndices,
        truncationIndices]];
858     (* These are the basis vectors for the truncated hamiltonian *)
859     PrintFun["Saving the truncated intermediate basis ..."];
860     truncatedIntermediateBasis = basisChanger[[All,truncationIndices
        ]];
861
862     PrintFun["Compiling a function for the truncated Hamiltonian ..."
        ];
863     (* Compile a function that will calculate the truncated
        Hamiltonian given the parameters in allVars, this is the function
        to be use in fitting *)
864     compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
865                                       Evaluate[N[Normal[
```

```
    truncatedIntermediateHam]]]];
866     (* Save the compiled function *)
867     PrintFun["Saving the compiled function for the truncated
        Hamiltonian and the truncatedIntermediateBasis..."];
868     Export[compileIntermediateFname, {compileIntermediateTruncatedHam
        , truncatedIntermediateBasis}];
869   )
870   ];
871
872   TheTruncatedAndSignedPathGenerator::usage = "This function puts
        together the necessary expression for defining a function which
        has as arguments all the symbolic values in varsMixedWithVals and
        which feeds to compileIntermediateTruncatedHam the arguments as
        given in varsMixedWithVals. varsMixedWithVals needs to respect the
         order of aruments expected by compileIntermediateTruncatedHam.
        Once the necessary template has been used this function then
        results in the definition of the function
        TheTruncatedAndSignedPath.";
873   TheTruncatedAndSignedPathGenerator[varsMixedWithVals_List]:=(
874     variableVars   = Select[varsMixedWithVals,Not[NumericQ[#]]&];
875     numQSignature = StringJoin[Riffle[(ToString[#]<>"_?NumericQ")&/
        @variableVars,", "]];
876     varWithValsSignature = StringJoin[Riffle[(ToString[#]<>"")&/
        @varsMixedWithVals,", "]];
877     funcString     = truncatedEnergyCostTemplate[<|"varsWithNumericQ"
        ->numQSignature,"varsMixedWithFixedVals" -> varWithValsSignature
        |>];
878     ClearAll[TheTruncatedAndSignedPath];
879     ToExpression[funcString]
880   );
881
882   (* We need to create a function call that has all the frozen
        parameters in place and all the active symbols unevaluated *)
883   (* find the indices of the activeVars to create the function
        signature *)
884   activeVarIndices = Flatten[Position[allVars,#]&/@activeVars];
885   (* we start from the numerical values in the current best*)
886   jobVars          = startingValues;
887   (* we then put back the symbols that should be unevaluated *)
888   jobVars[[activeVarIndices]] = activeVars;
889
890   oddsAndEnds["jobVars"] = jobVars;
891   (* calculate the constraints *)
892   constraints                = N[Constrainer[activeVars,1n]];
893   oddsAndEnds["constraints"] = constraints;
894   (* This is useful for the progress window *)
895   activeVarsString  = StringJoin[Riffle[ToString/@activeVars,", "]];
896   TheTruncatedAndSignedPathGenerator[jobVars];
897   stringPartialVars = ToString/@activeVars;
898
899   activeVarsWithRange = If[usingInitialRange,
900     MapIndexed[Flatten[{#1,
901         (1-Sign[startVarValues[[#2]]]*fractionalWidth) *
        startVarValues[[#2]],
902         (1+Sign[startVarValues[[#2]]]*fractionalWidth) *
```

```
    startVarValues[[#2]]
903         }]&, activeVars],
904     activeVars
905   ];
906
907   (* this is the template for the minimizer *)
908   solverTemplateNMini = StringTemplate["
909   numIter = 0;
910   sol = NMinimize[
911     Evaluate[
912       Join[{TheTruncatedAndSignedPath[`activeVarsString`]},
913         constraints
914       ]
915     ],
916     activeVarsWithRange,
917     AccuracyGoal   -> `accuracyGoal`,
918     MaxIterations  -> `maxIterations`,
919     Method->`Method`,
920     `Monitor`:>(
921       currentErr = TheTruncatedAndSignedPath[`activeVarsString`];
922       currentParams = activeVars;
923       numIter   += 1;
924       rmsHistory = AddToList[rmsHistory, currentErr, maxHistory,
      False];
925       paramSols  = AddToList[paramSols, activeVars, maxHistory, False
      ];
926       If[Not[runningInteractive],(
927         Print[numIter,\"/\",`maxIterations`];
928         Print[\"err    = \", ToString[NumberForm[Round[currentErr
      ,0.001],{Infinity,3}]]];
929         Print[\"params = \", ToString[NumberForm[Round[#,0.0001],{
      Infinity,4}]] &/@ currentParams];
930         )
931       ];
932     )
933   ]
934   "];
935   methodStringTemplate = StringTemplate["
936       {\"DifferentialEvolution\",
937         \"PostProcess\"   -> False,
938         \"ScalingFactor\" -> `DE:ScalingFactor`,
939         \"CrossProbability\" -> `DE:CrossProbability`,
940         \"RandomSeed\"    -> RandomInteger[{0,1000000}],
941         \"SearchPoints\"  -> `DE:SearchPoints`}"];
942   methodString = methodStringTemplate[<|
943     "DE:ScalingFactor" -> OptionValue["DE:ScalingFactor"],
944     "DE:CrossProbability" -> OptionValue["DE:CrossProbability"],
945     "DE:SearchPoints" -> OptionValue["DE:SearchPoints"]|>];
946   (* Evaluate the template *)
947   solverCode = solverTemplateNMini[<|
948     "accuracyGoal"  -> accuracyGoal,
949     "maxIterations" -> maxIterations,
950     "Method"->"{\"DifferentialEvolution\",
951                 \"PostProcess\"   -> False,
952                 \"ScalingFactor\" -> 0.6,
```

```
953                 \"CrossProbability\" -> 0.25,
954                 \"RandomSeed\"    -> RandomInteger[{0,1000000}],
955                 \"SearchPoints\"  -> Automatic}",
956   "Monitor"->"StepMonitor",
957   "activeVarsString"->activeVarsString|>
958  ];
959  threadHeaderTemplate = StringTemplate[ "(`idx`/`reps`) Fitting data
       for `ln` using `freeVars`."];
960  (* Find as many solutions as numReps *)
961  sols = Table[(
962    rmsHistory        = {};
963    paramSols         = {};
964    openNotebooks     = If[runningInteractive,
965              ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
     [],
966              {}];
967    If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
     ProgressView"],
968      ProgressNotebook["Basic"->False]
969    ];
970    If[Not[slackChan === None],
971    (
972      threadMessage = threadHeaderTemplate[<|"reps" -> numReps, "idx"
      -> rep, "ln" -> ln,
973         "freeVars" -> ToString[activeVars]|>];
974      threadTS = PostMessageToSlack[threadMessage, slackChan]["ts"];
975    )
976    ];
977    startTime = Now;
978    ToExpression[solverCode];
979
980    timeTaken = QuantityMagnitude[Now - startTime, "Seconds"];
981    Print["Took " <> ToString[timeTaken] <> "s"];
982    Print[sol];
983    bestError     = sol[[1]];
984    bestParams    = sol[[2]];
985    resultMessage = "sigma=" <> ToString[Round[bestError, 0.1]];
986    solAssoc = <|
987      "bestRMS"      -> bestError,
988      "solHistory"   -> rmsHistory,
989      "bestParams"   -> bestParams,
990      "paramHistory" -> paramSols,
991      "timeTaken/s"    -> timeTaken
992      |>;
993    solAssoc = Join[solAssoc,   oddsAndEnds];
994    logFname = LogSol[solAssoc, logFilePrefix];
995
996    If[Not[slackChan===None],(
997      PostMessageToSlack[resultMessage,   slackChan, "threadTS" ->
     threadTS];
998      PostFileToSlack[StringSplit[logFname,"/"][[-1]], logFname,
     slackChan, "threadTS" -> threadTS]
999    )
1000   ];
1001   solAssoc
```

```
    ),
    {rep,1,numReps}
  ];
  Return[sols];
);

ClassicalFit::usage="Classical[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.
1023
1024 The function returns an association with the following keys:
1025
1026 \"bestRMS\" which is the best \[Sigma] value found.
1027 \"bestParams\" which is the best set of parameters found.
1028 \"paramSols\" which is a list of the parameters during the stepping
       of the fitting algorithm.
1029 \"timeTaken/s\" which is the time taken to find the best fit.
1030 \"simplifier\" which is the simplifier used to simplify the
       Hamiltonian.
1031 \"excludeDataIndices\" as given in the input.
1032 \"starValues\" as given in the input.
1033
1034 \"freeIonSymbols\" which are the symbols used in the intermediate
       coupling basis.
1035 \"truncationEnergy\" which is the energy used to truncate the
       Hamiltonian.
1036 \"numE\" which is the number of electrons in the f^numE configuration
       .
1037 \"expData\" which is the experimental data used for fitting.
1038 \"problemVars\" which are the symbols considered for fitting
1039
1040 \"maxIterations\" which is the maximum number of iterations used by
       NMinimize.
1041 \"hamDim\" which is the dimension of the full Hamiltonian.
1042 \"allVars\" which are all the symbols defining the Hamiltonian under
       the aggregate simplifications.
1043 \"freeBies\" which are the free-ion parameters used to define the
       intermediate coupling basis.
1044 \"truncatedDim\" which is the dimension of the truncated Hamiltonian.
1045 \"compiledIntermediateFname\" the file name of the compiled function
       used for the truncated Hamiltonian.
1046
1047 \"fittedLevels\" which is the number of levels fitted for.
1048 \"actualSteps\" the number of steps that FindMiniminum actually took.
1049 \"solWithUncertainty\" which is a list of replacement rules whose
       left hand sides are symbols for the used parameters and whose's
       right hand sides are lists with the best fit value and the
       uncertainty in that value.
1050 \"rmsHistory\" which is a list of the \[Sigma] values found during
       the fitting.
1051 \"Appendix\" which is an association appended to the log file under
       the key \"Appendix\".
1052 \"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.
1053
```

```
1054 \"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.
1055 \"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.
1056
1057 The function admits the following options with default values:
1058   \"MaxHistory\" : determines how long the logs for the solver can be
        .
1059   \"MaxIterations\": determines the maximum number of iterations used
        by NMinimize.
1060   \"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\".
1061   \"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.
1062
1063   \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
       5.
1064   \"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.
1065   \"PrintFun\": the function used to print progress messages, the
       default is PrintTemporary.
1066
1067   \"SlackChannel\": name of the Slack channel to which to dump
       progress messaages, the default is None which disables this option
        .
1068   \"ProgressView\": whether or not a progress window will be opened
       to show the progress of the solver, the default is True.
1069   \"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
        startingValues.
1070   \"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.
1071
1072   \"AppendToLogFile\": an association appended to the log file under
       the key \"Appendix\".
1073   \"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,
  "SlackChannel"     -> None,
  "ProgressView"     -> True,
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "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, T11->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0},
    2 -> {
```

```
1122        T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
1123        T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
        ->0, T19->0,
1124        T2p->0
1125        },
1126      3  -> {},
1127      4  -> {},
1128      5  -> {},
1129      6  -> {},
1130      7  -> {},
1131      8  -> {},
1132      9  -> {},
1133      10 -> {},
1134      11 -> {},
1135      12 -> {
1136        T3->0, T4->0, T6->0, T7->0, T8->0,
1137        T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
        ->0, T19->0,
1138        T2p->0
1139        },
1140      13->{
1141        T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
1142        T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
        ->0, T19->0,
1143        T2p->0
1144        }
1145      |>,
1146      "FreeIonSymbols" -> {F0, F2, F4, F6, ζ}
1147  };
1148  ClassicalFit[numE_Integer, expData_List, excludeDataIndices_List,
        problemVars_List, startValues_Association, \[Sigma]exp_?NumericQ,
        constraints_List, OptionsPattern[]]:=
1149    (* Module[{accuracyGoal, activeVarIndices, activeVars,
        activeVarsString, activeVarsWithRange, allFreeEnergies,
        allFreeEnergiesSorted, allVars, allVarsVec,
        argsForEvalInsideOfTheIntermediateSystems,
        argsOfTheIntermediateEigensystems, aVar, aVarPosition, basis,
        basisChanger, basisChangerBlocks, bestError, bestParams, bestRMS,
        blockShifts, blockSizes, colIdx, compiledDiagonal,
        compiledIntermediateFname, constrainedProblemVars,
        constrainedProblemVarsList, covMat, currentRMS, degressOfFreedom,
        dependentVars, diagonalBlocks, diagonalScalarBlocks, diff,
        eigenEnergies, eigenvalueDispenserTemplate, eigenVectors,
        elevatedIntermediateEigensystems, endTime, fmSol, fmSolAssoc,
        fractionalWidth, freeBies, freeIenergiesAndMultiplets,
        freeionSymbols, fullHam, fullSolVec, funcString, ham, hamDim,
        hamEigenvaluesTemplate, hamString, hess, indepSolVecVec, indepVars
        , intermediateHam, isolationValues, jobVars, lin, linMat, ln,
        lnParams, logFilePrefix, logFname, magneticSimplifier,
        maxFreeEnergy, maxHistory, maxIterations, methodString,
        methodStringTemplate, minFreeEnergy, minpoly, modelSymbols,
        multipletAssignments, needlePosition, numBlocks, numQSignature,
        numReps, solCompendium, openNotebooks, ordering, othersFixed,
        otherSimplifier, p0, paramBest, paramSigma, perHam, polySols,
        presentDataIndices, PrintFun, problemVarsPositions, problemVarsQ,
```

```
    problemVarsQString, problemVarsVec, problemVarsWithStartValues,
    reducedModelSymbols, resultMessage, roundedTruncationEnergy,
    rowIdx, runningInteractive, shiftToggle, simplifier, slackChan,
    sol, solAssoc, sols, solWithUncertainty, sortedTruncationIndex,
    sqdiff, standardValues, starTime, startingValues, startTime,
    startVarValues, states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, thisPoly, threadHeaderTemplate
    , threadMessage, threadTS, timeTaken, totalVariance,
    truncadedFname, truncatedIntermediateBasis,
    truncatedIntermediateHam, truncationEnergy, truncationIndices,
    truncationUmbral, usingInitialRange, varHash, varIdx,
    varsWithConstants, varWithValsSignature, \[Lambda]0Vec, \[Lambda]
    exp}, *)
1150  Module[{},
1151  (
1152    solCompendium   = <||>;
1153    addShift       = OptionValue["AddConstantShift"];
1154    ln             = theLanthanides[[numE]];
1155    maxHistory     = OptionValue["MaxHistory"];
1156    maxIterations  = OptionValue["MaxIterations"];
1157    logFilePrefix  = If[OptionValue["FilePrefix"] == "",
1158                      ToString[theLanthanides[[numE]]],
1159                      OptionValue["FilePrefix"]
1160                    ];
1161    accuracyGoal  = OptionValue["AccuracyGoal"];
1162    slackChan     = OptionValue["SlackChannel"];
1163    PrintFun      = OptionValue["PrintFun"];
1164    freeIonSymbols       = OptionValue["FreeIonSymbols"];
1165    runningInteractive   = (Head[$ParentLink] === LinkObject);
1166    magneticSimplifier   = OptionValue["MagneticSimplifier"];
1167    magFieldSimplifier   = OptionValue["MagFieldSimplifier"];
1168    symmetrySimplifier   = OptionValue["SymmetrySimplifier"];
1169    otherSimplifier      = OptionValue["OtherSimplifier"];
1170    threeBodySimplifier  = If[Head[OptionValue["ThreeBodySimplifier"]]
       == Association,
1171                            OptionValue["ThreeBodySimplifier"][numE],
1172                            OptionValue["ThreeBodySimplifier"]
1173                          ];
1174
1175    truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic
       ,
1176      PrintFun["Truncation energy set to Automatic, using the maximum
       energy in the data ..."];
1177      Max[Select[First /@ expData, NumericQ[#] &]],
1178      OptionValue["TruncationEnergy"]
1179      ];
1180    PrintFun["Using a truncation energy of ", truncationEnergy, " K"
      ];
1181
1182    simplifier           = Join[magneticSimplifier,
1183                            magFieldSimplifier,
1184                            symmetrySimplifier,
1185                            threeBodySimplifier,
1186                            otherSimplifier];
```

```
1187
1188      PrintFun["Determining gaps in the data ..."];
1189      (* the indices that are numeric in expData whatever is non-
          numeric is assumed as a known gap *)
1190      presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
          , ___}]];
1191      (* some indices omitted here based on the excludeDataIndices
          argument *)
1192      presentDataIndices = Complement[presentDataIndices,
          excludeDataIndices];
1193
1194      hamDim                              = Binomial[14, numE];
1195      solCompendium["simplifier"]         = simplifier;
1196      solCompendium["excludeDataIndices"] = excludeDataIndices;
1197      solCompendium["startValues"]        = startValues;
1198      solCompendium["freeIonSymbols"]     = freeIonSymbols;
1199      solCompendium["truncationEnergy"]   = truncationEnergy;
1200      solCompendium["numE"]               = numE;
1201      solCompendium["expData"]            = expData;
1202      solCompendium["problemVars"]        = problemVars;
1203      solCompendium["maxIterations"]      = maxIterations;
1204      solCompendium["hamDim"]             = hamDim;
1205      solCompendium["constraints"]        = constraints;
1206      modelSymbols  = Sort[Select[paramSymbols, Not[MemberQ[Join[
          racahSymbols, chenSymbols,{t2Switch, \[Epsilon],gs}],#]]&]];
1207      (* remove the symbols that will be removed by the simplifier, no
          symbol should remain here that is not in the symbolic Hamiltonian
          *)
1208      reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
          simplifier],#]]&];
1209
1210      (* this is useful to understand what are the arguments of the
          truncated compiled Hamiltonian *)
1211      If[OptionValue["SignatureCheck"],
1212        (
1213          Print["Given the model parameters and the simplifying
          assumptions, the resultant model parameters are:"];
1214          Print[{reducedModelSymbols}];
1215          Print["Exiting ..."];
1216          Return[""];
1217        )
1218      ];
1219
1220      (* calculate the basis *)
1221      PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
1222      basis    = BasisLSJMJ[numE];
1223
1224      (* get the reference parameters from LaF3 *)
1225      PrintFun["Getting reference free-ion parameters for ",ln," using
          LaF3 ..."];
1226      lnParams = LoadParameters[ln];
1227      freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
           numE];
1228      (* a more explicit alias *)
1229      allVars           = reducedModelSymbols;
```

```
1230    numericConstraints = Association@Select[constraints, NumericQ
        [#[[2]]] &];
1231    standardValues     = allVars /. Join[lnParams, numericConstraints
        ];
1232    solCompendium["allVars"]  = allVars;
1233    solCompendium["freeBies"] = freeBies;
1234
1235    (* reload compiled version if found *)
1236    varHash                   = Hash[{numE, allVars, freeBies,
        truncationEnergy, simplifier}];
1237    compiledIntermediateFname = ln<>"-compiled-intermediate-truncated
        -ham-"<>ToString[varHash]<>".mx";
1238    compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
        compiledIntermediateFname}];
1239    solCompendium["compiledIntermediateFname"] =
        compiledIntermediateFname;
1240
1241    If[FileExistsQ[compiledIntermediateFname],
1242      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 ..."];
1243      PrintFun["Using : ", compiledIntermediateFname];
1244      {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
       Import[compiledIntermediateFname];,
1245    (
1246      (* grab the Hamiltonian preserving its block structure *)
1247      PrintFun["Assembling the Hamiltonian for fˆ",numE," keeping the
       block structure ..."];
1248      ham          = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
1249      (* apply the simplifier *)
1250      PrintFun["Simplifying using the aggregate set of simplification
       rules ..."];
1251      ham          = Map[ReplaceInSparseArray[#, simplifier]&, ham,
        {2}];
1252      PrintFun["Zeroing out every symbol in the Hamiltonian that is
        not a free-ion parameter ..."];
1253      (* Get the free ion symbols *)
1254      freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
       freeIonSymbols];
1255      (* Take the diagonal blocks for the intermediate analysis *)
1256      PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."
        ];
1257      diagonalBlocks       = Diagonal[ham];
1258      (* simplify them to only keep the free ion symbols *)
1259      PrintFun["Simplifying the diagonal blocks to only keep the free
       ion symbols ..."];
1260      diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
        ]&/@diagonalBlocks;
1261      (* these include the MJ quantum numbers, remove that *)
1262      PrintFun["Contracting the basis vectors by removing the MJ
        quantum numbers from the diagonal blocks ..."];
1263      diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
1264
1265      argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
```

```
    Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
1266    argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
    ToString[#]<>"v") & /@ freeIonSymbols,", "]];
1267    PrintFun["argsOfTheIntermediateEigensystems = ",
    argsOfTheIntermediateEigensystems];
1268    PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
    argsForEvalInsideOfTheIntermediateSystems];
1269    PrintFun["(if the following fails, it might help to see if the
    arguments of TheIntermediateEigensystems match the ones shown
    above)"];
1270
1271    (* 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 *)
1272    (* compile one function for each of the blocks *)
1273    PrintFun["Compiling functions for the diagonal blocks of the
    Hamiltonian ..."];
1274    compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N
    [Normal[#]]]]&/@diagonalScalarBlocks;
1275    (* use that to create a function that will calculate the free-
    ion eigensystem *)
1276    TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζv_
    ] := (
1277      theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&)/
    @compiledDiagonal;
1278      theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
1279      Js     = AllowedJ[numEv];
1280      basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
1281      (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
1282      elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2
    Js[[#2[[1]]]]+1]&, theIntermediateEigensystems];
1283      (* Identify a single MJ to keep *)
1284      pivot = If[EvenQ[numEv],0,-1/2];
1285      LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
1286      (* calculate the multiplet assignments that the intermediate
    basis eigenvectors have *)
1287      needlePosition = 0;
1288      multipletAssignments = Table[
1289        (
1290          J         = Js[[idx]];
1291          eigenVecs = theIntermediateEigensystems[[idx]][[2]];
1292          majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
1293          majorComponentIndices       += needlePosition;
1294          needlePosition              += Length[
    majorComponentIndices];
1295          majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
1296          (* All of the degenerate eigenvectors belong to the same
    multiplet*)
1297          elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments,2J+1];
1298          elevatedMultipletAssignments
```

```
1299          ),
1300        {idx, 1, Length[Js]}
1301        ];
1302        (* put together the multiplet assignments and the energies *)
1303        freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
1304        freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
1305        (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
1306        basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
1307        basisChanger = SparseArray[basisChanger];
1308        Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
1309      );
1310
1311      PrintFun["Calculating the intermediate eigensystems for ",ln,"
    using free-ion params from LaF3 ..."];
1312      (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
1313      {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
1314
1315      (* use that intermediate coupling basis to compile a function
    for the full Hamiltonian *)
1316      allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
1317      (* important that the intermediate coupling basis have attached
     energies, which make possible the truncation *)
1318      ordering = Ordering[allFreeEnergies];
1319      (* sort the free ion energies and determine which indices
    should be included in the truncation *)
1320      allFreeEnergiesSorted          = Sort[allFreeEnergies];
1321      {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
1322      (* determine the index at which the energy is equal or larger
    than the truncation energy *)
1323      sortedTruncationIndex = Which[
1324        truncationEnergy > (maxFreeEnergy-minFreeEnergy),
1325        hamDim,
1326        True,
1327        FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted
    ],x_/;x>truncationEnergy,{0},1][[1]]
1328      ];
1329      (* the actual energy at which the truncation is made *)
1330      roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
1331
1332      (* the indices that enact the truncation *)
1333      truncationIndices = ordering[[;;sortedTruncationIndex]];
1334      (* Return[{basisChanger, ham, truncationIndices}]; *)
1335      PrintFun["Computing the block structure of the change of basis
    array ..."];
```

```
1336        blockSizes = BlockArrayDimensionsArray[ham];
1337        basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
1338        blockShifts = First /@ Diagonal[blockSizes];
1339        numBlocks = Length[blockSizes];
1340        (* using the ham (with all the symbols) change the basis to the
             computed one *)
1341        PrintFun["Changing the basis of the Hamiltonian to the
            intermediate coupling basis ..."];
1342        (* intermediateHam            = Transpose[basisChanger].ham.
            basisChanger; *)
1343        (* Return[{basisChangerBlocks, ham}]; *)
1344        intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks];
1345        PrintFun["Distributing products inside of symbolic matrix
            elements to keep complexity in check ..."];
1346        Do[
1347          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
            intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1348          {rowIdx, 1, numBlocks},
1349          {colIdx, 1, numBlocks}
1350        ];
1351        intermediateHam = BlockMatrixMultiply[BlockTranspose[
            basisChangerBlocks],intermediateHam];
1352        PrintFun["Distributing products inside of symbolic matrix
            elements to keep complexity in check ..."];
1353        Do[
1354          intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
            intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1355          {rowIdx, 1, numBlocks},
1356          {colIdx, 1, numBlocks}
1357        ];
1358        (* using the truncation indices truncate that one *)
1359        PrintFun["Truncating the Hamiltonian ..."];
1360        truncatedIntermediateHam = TruncateBlockArray[intermediateHam,
            truncationIndices, blockShifts];
1361        (* these are the basis vectors for the truncated hamiltonian *)
1362        PrintFun["Saving the truncated intermediate basis ..."];
1363        truncatedIntermediateBasis = basisChanger[[All,
            truncationIndices]];
1364
1365        PrintFun["Compiling a function for the truncated Hamiltonian
            ..."];
1366        (* compile a function that will calculate the truncated
            Hamiltonian given the parameters in allVars, this is the function
            to be use in fitting *)
1367        compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
            Evaluate[truncatedIntermediateHam]];
1368        (* save the compiled function *)
1369        PrintFun["Saving the compiled function for the truncated
            Hamiltonian and the truncated intermediate basis ..."];
1370        Export[compiledIntermediateFname, {
            compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
1371      )
1372      ];
1373
1374      truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
```

```
1375    PrintFun["The truncated Hamiltonian has a dimension of ",
        truncationUmbral, "x", truncationUmbral, " ..."];
1376    presentDataIndices = Select[presentDataIndices, # <=
        truncationUmbral &];
1377    solCompendium["presentDataIndices"] = presentDataIndices;
1378
1379    (* the problemVars are the symbols that will be fitted for *)
1380
1381    PrintFun["Starting up the fitting process using the Levenberg-
        Marquardt method ..."];
1382    (* using the problemVars I need to create the argument list
        including _?NumericQ *)
1383    problemVarsQ = (ToString[#] <> "_?NumericQ") & /@ problemVars;
1384    problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
1385    (* we also need to have the padded arguments with the variables
        in the right position and the fixed values in the remaining ones
        *)
1386    problemVarsPositions = Position[allVars, #][[1, 1]] & /@
        problemVars;
1387    problemVarsString = StringJoin[Riffle[ToString /@ problemVars, ",
         "]];
1388    (* to feed parameters to the Hamiltonian, which includes all
        parameters, we need to form the rist set of arguments, with fixed
        values where needed, and the variables in the right position *)
1389    varsWithConstants = standardValues;
1390    varsWithConstants[[problemVarsPositions]] = problemVars;
1391    varsWithConstantsString = ToString[varsWithConstants];
1392
1393    (* this following function serves eigenvalues from the
        Hamiltonian, has memoization so it might grow to use a lot of RAM
        *)
1394    Clear[HamSortedEigenvalues];
1395    hamEigenvaluesTemplate = StringTemplate["
1396    HamSortedEigenvalues[`problemVarsQ`]:=(
1397      ham          = compileIntermediateTruncatedHam@@`
        varsWithConstants`;
1398      eigenValues = Sort@Eigenvalues@ham;
1399      eigenValues = eigenValues - Min[eigenValues];
1400      HamSortedEigenvalues[`problemVarsString`] = eigenValues;
1401      Return[eigenValues]
1402    )"];
1403    hamString = hamEigenvaluesTemplate[<|
1404        "problemVarsQ" -> problemVarsQString,
1405        "varsWithConstants" -> varsWithConstantsString,
1406        "problemVarsString" -> problemVarsString
1407        |>];
1408    ToExpression[hamString];
1409
1410    (* 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 *)
1411    eigenvalueDispenserTemplate = StringTemplate["
1412    PartialHamEigenvalues[`problemVarsQ`][i_]:=(
1413      eigenVals = HamSortedEigenvalues[`problemVarsString`];
1414      eigenVals[[i]]
```

```
    )
    "];
    eigenValueDispenserString =
      eigenvalueDispenserTemplate[<|
        "problemVarsQ"      -> problemVarsQString,
        "problemVarsString" -> problemVarsString
        |>];
    ToExpression[eigenValueDispenserString];

    PringFun["Determining the free variables after constraints ..."];
    constrainedProblemVars = (problemVars /. constraints);
    constrainedProblemVarsList = Variables[constrainedProblemVars];
    If[addShift,
      PrintFun["Adding a constant shift to the fitting parameters ...
    "];
      constrainedProblemVarsList = Append[constrainedProblemVarsList,
     \[Epsilon]]
    ];

    indepVars = Complement[pVars, #[[1]] & /@ constraints];
    stringPartialVars = ToString/@constrainedProblemVarsList;

    paramSols  = {};
    rmsHistory = {};
    steps      = 0;
    problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
    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["Starting the fitting process ..."];
    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;
        currentRMS = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues
    @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j,
    presentDataIndices}];
        currentRMS = Sqrt[currentRMS / 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 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]];
        Do[
          isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
          {dVar, dependentVars}
        ];
        perHam  = compileIntermediateTruncatedHam @@ isolationValues;
```

```
1508          lin     = FirstOrderPerturbation[Last /@ states, perHam];
1509          linMat  = Append[linMat, lin];
1510        ),
1511        {aVar, constrainedProblemVarsList[[;;tail]]}
1512      ];
1513      PrintFun["Removing the gradient of the ground state ..."];
1514      linMat = (# - #[[1]] & /@ linMat);
1515      PrintFun["Transposing derivative matrices into columns ..."];
1516      linMat  = Transpose[linMat];
1517
1518      PrintFun["Calculating the eigenvalue vector at solution ..."];
1519      \[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
1520      PrintFun["Putting together the experimental vector ..."];
1521      \[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
          ]]}];
1522      problemVarsVec = If[addShift,
1523        Transpose[{constrainedProblemVarsList[[;;-2]]}],
1524        Transpose[{constrainedProblemVarsList}]
1525      ];
1526      indepSolVecVec = Transpose[{indepSolVec}];
1527      PrintFun["Calculating the difference between eigenvalues at
          solution ..."];
1528      diff = If[linMat=={},
1529        (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
1530        (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
          presentDataIndices]].(problemVarsVec - indepSolVecVec)
1531      ];
1532      PrintFun["Calculating the sum of squares of differences around
          solution ... "];
1533      sqdiff         = Expand[(Transpose[diff] . diff)[[1, 1]]];
1534      PrintFun["Calculating the minimum (which should coincide with sol
           ..."];
1535      minpoly        = sqdiff /. AssociationThread[problemVars -> solVec
          ];
1536      fmSolAssoc     = Association[sol[[2]]];
1537      totalVariance = Length[presentDataIndices]*\[Sigma]exp^2;
1538      PrintFun["Calculating the uncertainty in the parameters ..."];
1539      solWithUncertainty = Table[
1540        (
1541          aVar        = constrainedProblemVarsList[[varIdx]];
1542          paramBest   = aVar /. fmSolAssoc;
1543          othersFixed = AssociationThread[Delete[
          constrainedProblemVarsList[[;;tail]], varIdx] -> Delete[
          indepSolVec, varIdx]];
1544          thisPoly    = sqdiff /. othersFixed;
1545          polySols    = Last /@ Last /@ Solve[thisPoly == minpoly + 1*
          totalVariance];
1546          polySols    = Select[polySols, Im[#] == 0 &];
1547          paramSigma  = Max[polySols] - Min[polySols];
1548          (aVar -> {paramBest, paramSigma})
1549        ),
1550      {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
1551      ];
1552      PrintFun["Calculating the covariance matrix ..."];
1553      hess = If[linmat=={},
```

```
      {{Infinity}},
      2 * Transpose[linMat[[presentDataIndices]]] . linMat[[
    presentDataIndices]]
    ];
    covMat  = If[linmat=={},
      {{0}},
      \[Sigma]exp^2 * Inverse[hess]
    ];
    bestRMS = Sqrt[minpoly / degressOfFreedom];
    solCompendium["truncatedDim"] = truncationUmbral;
    solCompendium["fittedLevels"] = Length[presentDataIndices];
    solCompendium["actualSteps"] = steps;
    solCompendium["bestRMS"]     = bestRMS;
    solCompendium["solWithUncertainty"] = solWithUncertainty;
    solCompendium["problemVars"] = problemVars;
    solCompendium["paramSols"]   = paramSols;
    solCompendium["rmsHistory"]  = rmsHistory;
    solCompendium["Appendix"]    = OptionValue["AppendToLogFile"];
    solCompendium["timeTaken/s"] = timeTaken;
    solCompendium["bestParams"]  = sol[[2]];
    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;
      )
    ];
    logFname = LogSol[solCompendium, logFilePrefix];
    Return[solCompendium];
  )
];


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" -> {},
"Pm" -> {},
"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}
    |>;
```

12.3 qonstants.m

This file has a few constants and conversion factors.

```
BeginPackage["qonstants`"];

(* Physical Constants*)
bohrRadius = 5.29177210903 * 10^-9;
ee         = 1.602176634 * 10^-19;

(* 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"

(* 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 *)

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

(* 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;
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[];
```

12.4 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.
211 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.
212 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.
213 ";
214 SpectrumPlot[lines_, widthToHeightAspect_, lineWidth_, opts :
    OptionsPattern[]] := Module[
215   {minWave, maxWave, height, fuzzyLines},
216 (
217   colorFun = OptionValue["SpectrumFunction"];
218   {minWave, maxWave} = MinMax[lines];
219   height     = (maxWave - minWave)/widthToHeightAspect;
220   fuzzyLines = Which[
221     NumberQ[lineWidth] && Length[OptionValue["Intensities"]] == 0,
222       FuzzyRectangle[#, lineWidth, 0, height, colorFun[#]] &/@
    lines,
223     Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]]
    == 0,
224       MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1]] &,
    {lines, lineWidth}],
225     NumberQ[lineWidth] && Length[OptionValue["Intensities"]] > 0,
226       MapThread[FuzzyRectangle[#1, lineWidth, 0, height, colorFun
    [#1], #2] &, {lines, OptionValue["Intensities"]}],
227     Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] >
     0,
228       MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1], #3]
     &, {lines, lineWidth, OptionValue["Intensities"]}]
229   ];
230   comments = Which[
231     Length[OptionValue["Comments"]] > 0,
232       MapThread[StringJoin[ToString[#1]<>" nm","\n",ToString[#2]]&,
233       {lines, OptionValue["Comments"]}],
234     Length[OptionValue["Comments"]] == 0,
235       ToString[#] <>" nm" & /@ lines,
236     True,
237     {}
238   ];
239   If[OptionValue["Tooltips"],
240     fuzzyLines = MapThread[Tooltip[#1, #2] &, {fuzzyLines, comments
    }];
241   ];
242   graphicsOpts = FilterRules[{opts}, Options[Graphics]];
243   Graphics[fuzzyLines,
244       graphicsOpts,
245       Background -> Black,
246       PlotRange -> {{minWave, maxWave}, {0, height}}]
247   )
248 ];
249
250 End[];
251
252 EndPackage[];
```

12.5 misc.m

This module includes a few functions useful for data-handling.

```
BeginPackage["misc`"];

ExportToH5;
FlattenBasis;
RecoverBasis;
FlowMatching;
SuperIdentity;
RobustMissingQ;
ReplaceDiagonal;

GreedyMatching;
HelperNotebook;
StochasticMatching;
ExtractSymbolNames;
GetModificationDate;
TextBasedProgressBar;
ToPythonSparseFunction;

FirstOrderPerturbation;
SecondOrderPerturbation;
RoundValueWithUncertainty;

ToPythonSymPyExpression;
RoundToSignificantFigures;
RobustMissingQ;

BlockMatrixMultiply;
BlockAndIndex;
TruncateBlockArray;
BlockArrayDimensionsArray;
ArrayBlocker;
BlockTranspose;
RemoveTrailingDigits;

Begin["`Private`"];

  RemoveTrailingDigits[s_String] := StringReplace[s,
    RegularExpression["\\d+$"] -> ""];

  BlockTranspose[anArray_]:=(
    Map[Transpose, Transpose[anArray], {2}]
  );

  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.";
  BlockMatrixMultiply[Amat_,Bmat_]:=Module[{rowIdx,colIdx,sumIdx},
  (
```

```
    Table[
      Sum[Amat[[rowIdx,sumIdx]].Bmat[[sumIdx,colIdx]],{sumIdx,1,
     Dimensions[Amat][[2]]}],
     {rowIdx,1,Dimensions[Amat][[1]]},
     {colIdx,1,Dimensions[Bmat][[2]]}
     ]
  )
  ];

  BlockAndIndex::usage="BlockAndIndex[blockSizes, index] takes a list
     of bin widths and index. The function return in which block the
     index would be, were the bins to be layed out from left to right.
     The function also returns the position within the bin in which it
     is accomodated. The function returns these two numbers as a list
     of two elements {blockIndex, blockSubIndex}";
  BlockAndIndex[blockSizes_List, index_Integer]:=Module[{
     accumulatedBlockSize,blockIndex, blockSubIndex},
  (
     accumulatedBlockSize = Accumulate[blockSizes];
     If[accumulatedBlockSize[[-1]]-index<0,
       Print["Index out of bounds"];
       Abort[]
     ];
     blockIndex    = Flatten[Position[accumulatedBlockSize-index,n_ /;
      n>=0]][[1]];
     blockSubIndex = Mod[index-accumulatedBlockSize[[blockIndex]],
     blockSizes[[blockIndex]],1];
     Return[{blockIndex,blockSubIndex}]
  )
  ];

  TruncateBlockArray::usage="TruncateBlockArray[blockArray,
     truncationIndices, blockWidths] takes a 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. They blocks in the given array
     may be SparseArray. This is equivalent to FlattenArray[blockArray
     ][truncationIndices, truncationIndices] but may be more efficient
     blockArray is sparse.";
  TruncateBlockArray[blockArray_,truncationIndices_,blockWidths_]:=
     Module[
  {truncatedArray,blockCol,blockRow,blockSubCol,blockSubRow},(
  truncatedArray = Table[
     {blockCol,blockSubCol} = BlockAndIndex[blockWidths,fullColIndex];
     {blockRow,blockSubRow} = BlockAndIndex[blockWidths,fullRowIndex];
     blockArray[[blockRow,blockCol]][[blockSubRow,blockSubCol]],
  {fullColIndex,truncationIndices},
  {fullRowIndex,truncationIndices}
  ];
  Return[truncatedArray]
  )
  ];

  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;
   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}
   ];
   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 + β perMatrix are to first order equal to \[Lambda] + \[
  Delta]_i β, where the \[Delta]_i are the returned values. This
  assuming that the eigenvalues are non-degenerate.";
FirstOrderPerturbation[eigenVectors_,
  perMatrix_] := (Diagonal[
  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.";
SecondOrderPerturbation[eigenValues_, eigenVectors_, perMatrix_] :=
   (
  dim = Length[perMatrix];
  eigenBras = Conjugate[eigenVectors];
  eigenKets = eigenVectors;
  matV = Abs[eigenBras . perMatrix . Transpose[eigenKets]]^2;
  OneOver[x_, y_] := If[x == y, 0, 1/(x - y)];
  eigenDiffs = Outer[OneOver, eigenValues, eigenValues, 1];
  pProduct = Transpose[eigenDiffs]*matV;
  Return[2*(Total /@ Transpose[pProduct])];
  )

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.";
SuperIdentity[args___] := {args};

FlattenBasis::usage="FlattenBasis[basis] takes a basis in the
  standard representation and separates out the strings that
  describe the LS part of the labels and the additional numbers that
   define the values of J MJ and MI. It returns a list with two
  elements {flatbasisLS, flatbasisNums}. This is useful for saving
  the basis to an h5 file where the strings and numbers need to be
  separated.";
FlattenBasis[basis_] := Module[{flatbasis, flatbasisLS,
  flatbasisNums},
  (
    flatbasis = Flatten[basis];
    flatbasisLS = flatbasis[[1 ;; ;; 4]];
    flatbasisNums = Select[flatbasis, Not[StringQ[#]] &];
```

```
198       Return[{flatbasisLS, flatbasisNums}]
199       )
200     ];
201
202   RecoverBasis::usage="RecoverBasis[{flatBasisLS, flatbasisNums}]
        takes the output of FlattenBasis and returns the original basis.
        The input is a list with two elements {flatbasisLS, flatbasisNums
        }.";
203   RecoverBasis[{flatbasisLS_, flatbasisNums_}] := Module[{recBasis},
204     (
205     recBasis = {{{#[[1]], #[[2]]}, #[[3]]}, #[[4]]} & /@ (Flatten /@
206         Transpose[{flatbasisLS,
207           Partition[Round[2*#]/2 & /@ flatbasisNums, 3]}]);
208     Return[recBasis];
209     )
210     ]
211
212   ExtractSymbolNames[expr_Hold] := Module[
213     {strSymbols},
214     strSymbols = ToString[expr, InputForm];
215     StringCases[strSymbols, RegularExpression["\\w+"]][[2 ;;]]
216     ]
217
218   ExportToH5::usage =
219     "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
      with mixes numbers and string will fail, but an Association with
      mixed values exports ok. Do give it a try.
220     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.";
221   Options[ExportToH5] = {"Overwrite" -> True};
222   ExportToH5[fname_String, symbols_Hold, OptionsPattern[]] := (
223     If[And[FileExistsQ[fname], OptionValue["Overwrite"]],
224     (
225       Print["File already exists, overwriting ..."];
226       DeleteFile[fname];
227       )
228     ];
229     symbolNames = ExtractSymbolNames[symbols];
230     Do[(Print[symbolName];
231       Export[fname, ToExpression[symbolName], {"Datasets", symbolName
      },
232       OverwriteTarget -> "Append"]
233       ), {symbolName, symbolNames}]
234     )
235
236   GreedyMatching::usage="GreedyMatching[aList, bList] returns a list
      of pairs of elements from aList and bList that are closest to each
       other, this is returned in a list together with a mapping of
      indices from the aList to those in bList to which they were
      matched. The option \"alistLabels\" can be used to specify labels
```

```
    for the elements in aList. The option \"blistLabels\" can be used
    to specify labels for the elements in bList. If these options are
    used, the function returns a list with three elements the pairs of
     matched elements, the pairs of corresponding matched labels, and
    the mapping of indices.";
237 Options[GreedyMatching] = {
238     "alistLabels" -> {},
239     "blistLabels" -> {}};
240 GreedyMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
241   aValues = aValues0,
242   bValues = bValues0,
243   bValuesOriginal = bValues0,
244   bestLabels, bestMatches,
245   bestLabel, aElement, givenLabels,
246   aLabels, aLabel,
247   diffs, minDiff,
248   bLabels,
249   minDiffPosition, bestMatch},
250   (
251   aLabels     = OptionValue["alistLabels"];
252   bLabels     = OptionValue["blistLabels"];
253   bestMatches = {};
254   bestLabels  = {};
255   givenLabels = (Length[aLabels] > 0);
256   Do[
257     (
258     aElement        = aValues[[idx]];
259     diffs           = Abs[bValues - aElement];
260     minDiff         = Min[diffs];
261     minDiffPosition = Position[diffs, minDiff][[1, 1]];
262     bestMatch       = bValues[[minDiffPosition]];
263     bestMatches     = Append[bestMatches, {aElement, bestMatch}];
264     If[givenLabels,
265       (
266       aLabel     = aLabels[[idx]];
267       bestLabel  = bLabels[[minDiffPosition]];
268       bestLabels = Append[bestLabels, {aLabel, bestLabel}];
269       bLabels    = Drop[bLabels, {minDiffPosition}];
270       )
271       ];
272     bValues = Drop[bValues, {minDiffPosition}];
273     If[Length[bValues] == 0, Break[]];
274     ),
275     {idx, 1, Length[aValues]}
276     ];
277   pairedIndices = MapIndexed[{#2[[1]], Position[bValuesOriginal,
    #1[[2]]][[1, 1]]} &, bestMatches];
278   If[givenLabels,
279     Return[{bestMatches, bestLabels, pairedIndices}],
280     Return[{bestMatches, pairedIndices}]
281     ]
282   )
283   ]
284
285 StochasticMatching::usage="StochasticMatching[aValues, bValues]
```

```
    finds a better assignment by randomly shuffling the elements of
    aValues and then applying the greedy assignment algorithm. The
    function prints what is the range of total absolute differences
    found during shuffling, the standard deviation of all of them, and
     the number of shuffles that were attempted. The option \"
    alistLabels\" can be used to specify labels for the elements in
    aValues. The option \"blistLabels\" can be used to specify labels
    for the elements in bValues. If these options are used, the
    function returns a list with three elements the pairs of matched
    elements, the pairs of corresponding matched labels, and the
    mapping of indices.";
286 Options[StochasticMatching] = {"alistLabels" -> {},
287   "blistLabels" -> {}};
288 StochasticMatching[aValues0_, bValues0_, numShuffles_ : 200,
    OptionsPattern[]] := Module[{
289   aValues = aValues0,
290   bValues = bValues0,
291   matchingLabels, ranger, matches, noShuff, bestMatch, highestCost,
     lowestCost, dev, sorter, bestValues,
292   pairedIndices, bestLabels, matchedIndices, shuffler
293   },
294   (
295   matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
296   ranger = Range[1, Length[aValues]];
297   matches = If[Not[matchingLabels], (
298       Table[(
299         shuffler = If[i == 1, ranger, RandomSample[ranger]];
300         {bestValues, matchedIndices} =
301         GreedyMatching[aValues[[shuffler]], bValues];
302         cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
303         {cost, {bestValues, matchedIndices}}
304         ), {i, 1, numShuffles}]
305       ),
306     Table[(
307       shuffler = If[i == 1, ranger, RandomSample[ranger]];
308       {bestValues, bestLabels, matchedIndices} =
309         GreedyMatching[aValues[[shuffler]], bValues,
310         "alistLabels" -> OptionValue["alistLabels"][[shuffler]],
311         "blistLabels" -> OptionValue["blistLabels"]];
312       cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
313       {cost, {bestValues, bestLabels, matchedIndices}}
314       ), {i, 1, numShuffles}]
315     ];
316   noShuff = matches[[1, 1]];
317   matches = SortBy[matches, First];
318   bestMatch = matches[[1, 2]];
319   highestCost = matches[[-1, 1]];
320   lowestCost = matches[[1, 1]];
321   dev = StandardDeviation[First /@ matches];
322   Print[lowestCost, " <-> ", highestCost, " | \[Sigma]=", dev,
323     " | N=", numShuffles, " | null=", noShuff];
324   If[matchingLabels,
325     (
326     {bestValues, bestLabels, matchedIndices} = bestMatch;
327     sorter = Ordering[First /@ bestValues];
```

```
328       bestValues = bestValues[[sorter]];
329       bestLabels = bestLabels[[sorter]];
330       pairedIndices =
331         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
332         bestValues];
333       Return[{bestValues, bestLabels, pairedIndices}]
334       ),
335       (
336       {bestValues, matchedIndices} = bestMatch;
337       sorter = Ordering[First /@ bestValues];
338       bestValues = bestValues[[sorter]];
339       pairedIndices =
340         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
341         bestValues];
342       Return[{bestValues, pairedIndices}]
343       )
344       ];
345     )
346     ]
347
348   FlowMatching::usage="FlowMatching[aList, bList] returns a list of
      pairs of elements from aList and bList that are closest to each
      other, this is returned in a list together with a mapping of
      indices from the aList to those in bList to which they were
      matched. The option \"alistLabels\" can be used to specify labels
      for the elements in aList. The option \"blistLabels\" can be used
      to specify labels for the elements in bList. If these options are
      used, the function returns a list with three elements the pairs of
       matched elements, the pairs of corresponding matched labels, and
      the mapping of indices. This is basically a wrapper around
      Mathematica's FindMinimumCostFlow function. By default the option
      \"notMatched\" is zero, and this means that all elements of aList
      must be matched to elements of bList. If this is not the case, the
       option \"notMatched\" can be used to specify how many elements of
       aList can be left unmatched. By default the cost function is Abs
      [#1-#2]&, but this can be changed with the option \"CostFun\",
      this function needs to take two arguments.";
349   Options[FlowMatching] = {"alistLabels" -> {}, "blistLabels" -> {},
      "notMatched" -> 0, "CostFun"-> (Abs[#1-#2] &)};
350   FlowMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
351     aValues = aValues0, bValues = bValues0, edgesSourceToA,
      capacitySourceToA, nA, nB,
352     costSourceToA, midLayer, midLayerEdges, midCapacities,
353     midCosts, edgesBtoSink, capacityBtoSink, costBtoSink,
354     allCapacities, allCosts, allEdges, graph,
355     flow, bestValues, bestLabels, cFun,
356     aLabels, bLabels, pairedIndices, matchingLabels},
357     (
358     matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
359     aLabels = OptionValue["alistLabels"];
360     bLabels = OptionValue["blistLabels"];
361     cFun    = OptionValue["CostFun"];
362     nA      = Length[aValues];
363     nB      = Length[bValues];
364     (*Build up the edges costs and capacities*)
```

```
    (*From source to the nodes representing the values of the first \
list*)
    edgesSourceToA     = ("source" \[DirectedEdge] {"A", #}) & /@
    Range[1, nA];
    capacitySourceToA = ConstantArray[1, nA];
    costSourceToA      = ConstantArray[0, nA];

    (*From all the elements of A to all the elements of B*)
    midLayer = Table[{{"A", i} \[DirectedEdge] ({"B", j}), 1, cFun[
    aValues[[i]], bValues[[j]]]}, {i, 1, nA}, {j, 1, nB}];
    midLayer = Flatten[midLayer, 1];
    {midLayerEdges, midCapacities, midCosts} = Transpose[midLayer];

    (*From the elements of B to the sink*)
    edgesBtoSink     = ({"B", #} \[DirectedEdge] "sink") & /@ Range[1,
     nB];
    capacityBtoSink = ConstantArray[1, nB];
    costBtoSink      = ConstantArray[0, nB];

    (*Put it all together*)
    allCapacities = Join[capacitySourceToA, midCapacities,
    capacityBtoSink];
    allCosts       = Join[costSourceToA, midCosts, costBtoSink];
    allEdges       = Join[edgesSourceToA, midLayerEdges, edgesBtoSink
    ];
    graph          = Graph[allEdges, EdgeCapacity -> allCapacities,
      EdgeCost -> allCosts];

    (*Solve it*)
    flow           = FindMinimumCostFlow[graph, "source", "sink", nA -
     OptionValue["notMatched"], "OptimumFlowData"];
    (*Collect the pairs of matched indices*)
    pairedIndices = Select[flow["EdgeList"], And[Not[#[[1]] === "
    source"], Not[#[[2]] === "sink"]] &];
    pairedIndices = {#[[1, 2]], #[[2, 2]]} & /@ pairedIndices;
    (*Collect the pairs of matched values*)
    bestValues     = {aValues[[#[[1]]]], bValues[[#[[2]]]]} & /@
    pairedIndices;
    (*Account for having been given labels*)
    If[matchingLabels,
      (
      bestLabels = {aLabels[[#[[1]]]], bLabels[[#[[2]]]]} & /@
    pairedIndices;
      Return[{bestValues, bestLabels, pairedIndices}]
      ),
      (
      Return[{bestValues, pairedIndices}]
      )
      ];
    )
    ]

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.";
HelperNotebook[nbName_:"OUT"] :=
Module[{screenDims, screenWidth, screenHeight, nbWidth, leftMargin,
  PrintToOutputNb}, (
  screenDims =
    SystemInformation["Devices", "ScreenInformation"][[1, 2, 2]];
  screenWidth = screenDims[[1, 2]];
  screenHeight = screenDims[[2, 2]];
  nbWidth = Round[screenWidth/3];
  leftMargin = screenWidth - nbWidth;
  outputNb = CreateDocument[{}, WindowTitle -> nbName,
    WindowMargins -> {{leftMargin, Automatic}, {Automatic,
        Automatic}}, WindowSize -> {nbWidth, screenHeight}];
  PrintToOutputNb[text_] :=
    (
        SelectionMove[outputNb, After, Notebook];
        NotebookWrite[outputNb, Cell[BoxData[ToBoxes[text]], "
  Output"]];
    );
  Return[PrintToOutputNb]
  )
  ]

GetModificationDate::usage="GetModificationDate[fname] returns the
  modification date of the given file.";
GetModificationDate[theFileName_] := FileDate[theFileName, "
  Modification"];

(*Helper function to convert Mathematica expressions to standard
  form*)
StandardFormExpression[expr0_] := Module[{expr=expr0}, ToString[
  expr, InputForm]];

(*Helper function to translate to Python/SymPy expressions*)
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."
  ;
ToPythonSymPyExpression[expr0_] := Module[{standardForm, expr=expr0
  },
  standardForm = StandardFormExpression[expr];
  StringReplace[standardForm, {
    "Power[" -> "Pow(",
    "Sqrt[" -> "sqrt(",
    "[" -> "(",
    "]" -> ")",
    "\\" -> "",
    "I" -> "1j",
    (*Remove special Mathematica backslashes*)
    "/" -> "/" (*Ensure division is represented with a slash*)}]];

ToPythonSparseFunction[sparseArray_SparseArray, funName_] :=
  Module[{data, rowPointers, columnIndices, dimensions, pyCode,
  vars,
```

```
      varList, dataPyList,
      colIndicesPyList},(*Extract unique symbolic variables from the
    \
  SparseArray*)
    vars = Union[Cases[Normal[sparseArray], _Symbol, Infinity]];
    varList = StringRiffle[ToString /@ vars, ", "];
    (*varList=ToPythonSymPyExpression/@varList;*)
    (*Convert data to SymPy compatible strings*)
    dataPyList =
      StringRiffle[
      ToPythonSymPyExpression /@ Normal[sparseArray["NonzeroValues"
    ]],
      ", "];
    colIndicesPyList =
      StringRiffle[
      ToPythonSymPyExpression /@ (Flatten[
          Normal[sparseArray["ColumnIndices"]] - 1]), ", "];
    (*Extract sparse array properties*)
    rowPointers = Normal[sparseArray["RowPointers"]];
    dimensions = Dimensions[sparseArray];
    (*Create Python code string*)pyCode = StringJoin[
      "#!/usr/bin/env python3\n\n",
      "from scipy.sparse import csr_matrix\n",
      "from sympy import *\n",
      "import numpy as np\n",
      "\n",
      "sqrt = np.sqrt\n",
      "\n",
      "def ", funName, "(",
      varList,
      "):\n",
      "    data = np.array([", dataPyList, "])\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)"];
    pyCode
    ];

End[];
EndPackage[];
```

12.6 qalculations.m

This script encapsulates example calculations in which the level structure and magnetic dipole transitions are calculated for the lanthanide ions in lanthanum fluoride.

```
Needs["qlanth`"];
Needs["misc`"];
Needs["qplotter`"];
```

```
Needs["qonstants`"];
LoadCarnall[];

workDir = DirectoryName[$InputFileName];

FastIonSolverLaF3::usage = "This function solves the energy levels of the given trivalent lanthanide in LaF3. The values for the parameters in the Hamiltonian are  taken from the values quoted by Carnall. It can use precomputed symbolic matrices for the Hamiltonian if they have been loaded already and defined as a value of symbolicHamiltonians.

The function returns a list with nine elements {rmsDifference, carnallEnergies, eigenEnergies, ln, carnallAssignments, simplerStateLabels, eigensys, basis, truncatedStates}. The elements of the list are as follows:

1. rmsDifference is the root mean squared difference between the calculated values and those quoted by Carnall;
2. carnallEnergies are the quoted calculated energies from Carnall used for comparison;
3. eigenEnergies are the calculated energies (in the case of an odd number of electrons the Kramers degeneracy may have been removed from this list according to the option \"Remove Kramers\");
4. ln is simply a string labelling the corresponding lanthanide;
5. carnallAssignments is a list of strings providing the multiplet assignments that Carnall assumed;
6. simplerStateLabels is a list of strings providing the multiplet assignments that this function assumes;
7. eigensys is a list of tuples where the first element is the energy corresponding to the eigenvector given as the second element (in the case of an odd number of electrons the Kramers degeneracy may have been removed from this list according to the option \"Remove Kramers\");
8. basis is a list that specifies the basis in which the Hamiltonian was constructed and diagonalized, equal to BasisLSJMJ[numE];
9. truncatedStates is the same as eigensys but with the truncated eigenvectors so that the total propability add up to at least eigenstateTruncationProbability.

This function admits the following options:
  - \"MakeNotebook\" -> True or False. If True, a notebook with a summary of the data is created. Default is True.
  - \"NotebookSave\" -> True or False. If True, the results notebook is saved automatically. Default is True.
  - \"eigenstateTruncationProbability\" -> 0.9. The probability sum of the truncated eigenvectors. Default is 0.9.
  - \"Include spin-spin\" -> True or False. If True, the spin-spin contribution to the magnetic interactions is included. Default is True.
  - \"Max Eigenstates in Table\" -> 100. The maximum number of eigenstates to be shown in the table shown in the results notebook. Default is 100.
  - \"Sparse\" -> True or False. If True, the numerical Hamiltonian is kept in sparse form. Default is True.
```

```
  - \"PrintFun\" -> Print, PrintTemporary, or other to serve as a
    printer for progress messages. Default is Print.
  - \"SaveData\" -> True or False. If True, the resulting data is
    saved to disk. Default is True.
  - \"ParamOverride\". An association that can override parameters in
     the Hamiltonian. Default is <||>. This override cannot change the
     inclusion or exclusion of the spin-spin contribution to the
    magnetic interactions, for this purpose use the option \"Include
    spin-spin\".
  - \"Append to Filename\" -> \"\". A string to append to the
    filename of the saved notebook and data files. Default is \"\".
  - \"Remove Kramers\" -> True or False. If True, the Kramers
    degeneracy is removed from the eigenstates. Default is True.
  - \"OutputDirectory\" -> \"calcs\". The directory where the output
    files are saved. Default is \"calcs\".
  - \"Explorer\" -> True or False. If True, the energy level diagram
    is interactive. Default is False.
";
Options[FastIonSolverLaF3] = {
  "MakeNotebook"      -> True,
  "NotebookSave"      -> True,
  "Include spin-spin" -> True,
  "eigenstateTruncationProbability" -> 0.9,
  "Max Eigenstates in Table" -> 100,
  "Sparse" -> True,
  "PrintFun" -> Print,
  "SaveData" -> True,
  "ParamOverride" -> <||>,
  "Append to Filename" -> "",
  "Remove Kramers" -> True,
  "OutputDirectory" -> "calcs",
  "Explorer" -> False
};
FastIonSolverLaF3[numE_, OptionsPattern[]] := Module[
  {makeNotebook, eigenstateTruncationProbability, host,
  ln, terms, termNames, carnallEnergies, eigenEnergies,
    simplerStateLabels,
  eigensys, basis, assignmentMatches, stateLabels, carnallAssignments
    },
  (
    PrintFun = OptionValue["PrintFun"];
    makeNotebook = OptionValue["MakeNotebook"];
    eigenstateTruncationProbability = OptionValue["
    eigenstateTruncationProbability"];
    maxStatesInTable = OptionValue["Max Eigenstates in Table"];
    Duplicator[aList_] := Flatten[{#, #} & /@ aList];
    host = "LaF3";
    ParamOverride = OptionValue["ParamOverride"];
    ln = theLanthanides[[numE]];
    terms = AllowedNKSLJTerms[Min[numE, 14 - numE]];
    termNames = First /@ terms;
    (* For labeling the states, the degeneracy in some of the terms
    is elided *)
    PrintFun["> Calculating simpler term labels ..."];
    termSimplifier = Table[termN -> If[StringLength[termN] == 3,
```

```
71        StringTake[termN, {1, 2}],
72        termN
73        ],
74        {termN, termNames}
75      ];
76
77      (*Load the parameters from Carnall*)
78      PrintFun["> Loading the fit parameters from Carnall ..."];
79      params = LoadParameters[ln, "Free Ion" -> False];
80      If[numE>7,
81        (
82          PrintFun["> Conjugating the parameters accounting for the
    hole-particle equivalence ..."];
83          params = HoleElectronConjugation[params];
84          params[t2Switch] = 0;
85        ),
86        params[t2Switch] = 1;
87      ];
88
89      (* Apply the parameter override *)
90      Do[params[key] = ParamOverride[key],
91        {key, Keys[ParamOverride]}
92      ];
93
94      (* Import the symbolic Hamiltonian *)
95      PrintFun["> Loading the symbolic Hamiltonian for this
    configuration ..."];
96      startTime = Now;
97      numH = 14 - numE;
98      numEH = Min[numE, numH];
99      C2vsimplifier = {
100       B12 -> 0, B14 -> 0, B16 -> 0, B34 -> 0, B36 -> 0, B56 -> 0,
101       S12 -> 0, S14 -> 0, S16 -> 0, S22 -> 0, S24 -> 0, S26 -> 0, S34
     -> 0, S36 -> 0,S44 -> 0, S46 -> 0, S56 -> 0, S66 -> 0,
102       T11p -> 0, T11 -> 0, T12 -> 0, T14 -> 0, T15 -> 0, T16 -> 0,
    T18 -> 0, T17 -> 0, T19 -> 0};
103     (* If the necessary symbolicHamiltonian is define load if not
    make it *)
104     simpleHam = If[
105       ValueQ[symbolicHamiltonians[numEH]],
106       symbolicHamiltonians[numEH],
107       SimplerSymbolicHamMatrix[numE, C2vsimplifier, "
    PrependToFilename" -> "C2v-", "Overwrite" -> False]
108     ];
109     endTime  = Now;
110     loadTime = QuantityMagnitude[endTime - startTime, "Seconds"];
111     PrintFun[">> Loading the symbolic Hamiltonian took ", loadTime, "
     seconds."];
112
113     (*Enforce the override to the spin-spin contribution to the
    magnetic interactions*)
114     params[\[Sigma]SS] = If[OptionValue["Include spin-spin"], 1, 0];
115
116     (*Everything that is not given is set to zero*)
117     params = ParamPad[params, "Print" -> False];
```

```
    PrintFun[params];
    numHam = ReplaceInSparseArray[simpleHam, params];
    If[Not[OptionValue["Sparse"]],
        numHam = Normal[numHam]
    ];
    PrintFun["> Calculating the SLJ basis ..."];
    basis = BasisLSJMJ[numE];

    (* Eigensolver *)
    PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
    startTime = Now;
    eigensys  = Eigensystem[numHam];
    endTime   = Now;
    diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"];
    PrintFun[">> Diagonalization took ", diagonalTime, " seconds."];
    eigensys = Chop[eigensys];
    eigensys = Transpose[eigensys];

    (* Shift the baseline energy *)
    eigensys = ShiftedLevels[eigensys];
    (* Sort according to energy *)
    eigensys = SortBy[eigensys, First];
    (* Grab just the energies *)
    eigenEnergies = First /@ eigensys;

    (* Energies are doubly degenerate in the case of odd number of
    electrons, keep only one *)
    If[And[OddQ[numE], OptionValue["Remove Kramers"]],
        (
        PrintFun["> Since there's an odd number of electrons energies
     come in pairs, taking just one for each pair ..."];
        eigenEnergies = eigenEnergies[[;; ;; 2]];
        )
    ];

    (* Compare against the data quoted by Bill Carnall *)
    PrintFun["> Comparing against the data from Carnall ..."];
    mainKey            = StringTemplate["appendix:`Ln`:Association"
    ][<|"Ln" -> ln|>];
    lnData             = Carnall[mainKey];
    carnalKeys         = lnData // Keys;
    repetitions        = Length[lnData[#]["Calc (1/cm)"]] & /@
    carnalKeys;
    carnallAssignments = First /@ Carnall["appendix:" <> ln <> ":
    RawTable"];
    carnallAssignments = Select[carnallAssignments, Not[# === ""] &];
    carnalKey          = StringTemplate["appendix:`Ln`:Calculated"
    ][<|"Ln" -> ln|>];
    carnallEnergies    = Carnall[carnalKey];

    If[And[OddQ[numE], Not[OptionValue["Remove Kramers"]]],
    (
      PrintFun[">> The number of eigenstates and the number of quoted
     states don't match, removing the last state ..."];
      carnallAssignments = Duplicator[carnallAssignments];
```

```
    carnallEnergies    = Duplicator[carnallEnergies];
  )
  ];

  (* For the difference take as many energies as quoted by Bill *)
  eigenEnergies = eigenEnergies + carnallEnergies[[1]];
  diffs = Sort[eigenEnergies][[;; Length[carnallEnergies]]] -
 carnallEnergies;
  (* Remove the differences where the appendix tables have elided
 values*)
  rmsDifference = Sqrt[Mean[(Select[diffs, FreeQ[#, Missing[]] &])
 ^2]];
  titleTemplate = StringTemplate[
    "Energy Level Diagram of \!\(\*SuperscriptBox[\(`ion`\),
 \(\(3\)\(+\)\)]\)"];
  title = titleTemplate[<|"ion" -> In|>];
  parsedStates = ParseStates[eigensys, basis];
  If[And[OddQ[numE],OptionValue["Remove Kramers"]],
    parsedStates = parsedStates[[;; ;; 2]]
  ];

  stateLabels = #[[-1]] & /@ parsedStates;
  simplerStateLabels = ((#[[2]] /. termSimplifier) <> ToString
 [#[[3]], InputForm]) & /@ parsedStates;

  PrintFun[">> Truncating eigenvectors to given probability ..."];
  startTime = Now;
  truncatedStates = ParseStatesByProbabilitySum[eigensys, basis,
    eigenstateTruncationProbability,
    0.01];
  endTime = Now;
  truncationTime = QuantityMagnitude[endTime - startTime, "Seconds"
 ];
  PrintFun[">>> Truncation took ", truncationTime, " seconds."];

  If[makeNotebook,
  (
    PrintFun["> Putting together results in a notebook ..."];
    energyDiagram = Framed[
      EnergyLevelDiagram[eigensys, "Title" -> title,
      "Explorer" -> OptionValue["Explorer"],
      "Background" -> White]
      , Background -> White, FrameMargins -> 50];
    appToFname = OptionValue["Append to Filename"];
    PrintFun[">> Comparing the term assignments between qlanth and
 Carnall ..."];
    AssignmentMatchFunc = Which[
      StringContainsQ[#[[1]], #[[2]]],
      "\[Checkmark]",
      True,
      "X"] &;
    assignmentMatches = AssignmentMatchFunc /@ Transpose[{
 carnallAssignments, simplerStateLabels[[;; Length[
 carnallAssignments]]]}];
    assignmentMatches = {{"\[Checkmark]",
```

```
212         Count[assignmentMatches, "\[Checkmark]"]}, {"X",
213         Count[assignmentMatches, "X"]}};
214       labelComparison = (AssignmentMatchFunc /@ Transpose[{
    carnallAssignments, simplerStateLabels[[;; Length[
    carnallAssignments]]]}]);
215       labelComparison = PadRight[labelComparison, Length[
    simplerStateLabels], "-"];
216
217       statesTable = Grid[Prepend[{Round[#[[1]]], #[[2]]} & /@
218           truncatedStates[[;;Min[Length[eigensys],maxStatesInTable
    ]]], {"Energy/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
219         "\[Psi]"}], Frame -> All, Spacings -> {2, 2},
220         FrameStyle -> Blue,
221         Dividers -> {{False, True, False}, {True, True}}];
222       DefaultIfMissing[expr_]:= If[FreeQ[expr, Missing[]], expr,"NA"
    ];
223       PrintFun[">> Rounding the energy differences for table
    presentation ..."];
224       roundedDiffs = Round[diffs, 0.1];
225       roundedDiffs = PadRight[roundedDiffs, Length[simplerStateLabels
    ], "-"];
226       roundedDiffs = DefaultIfMissing /@ roundedDiffs;
227       diffs = PadRight[diffs, Length[simplerStateLabels], "-"];
228       diffs = DefaultIfMissing /@ diffs;
229       diffTableData = Transpose[{simplerStateLabels, eigenEnergies,
230         labelComparison,
231         PadRight[carnallAssignments, Length[simplerStateLabels], "-"
    ],
232         DefaultIfMissing/@PadRight[carnallEnergies, Length[
    simplerStateLabels], "-"],
233         roundedDiffs}
234       ];
235       diffTable = TableForm[diffTableData,
236         TableHeadings -> {None, {"qlanth",
237         "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)", "", "Carnall",
238         "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
239         "\[CapitalDelta]E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)"}}
240       ];
241
242       diffs = Sort[eigenEnergies][[;; Length[carnallEnergies]]] -
    carnallEnergies;
243       notBad = FreeQ[#,Missing[]]&/@diffs;
244       diffs = Pick[diffs, notBad];
245       (* diffHistogram = Histogram[diffs,
246         Frame -> True,
247         ImageSize -> 800,
248         AspectRatio -> 1/3, FrameStyle -> Directive[16],
249         FrameLabel -> {"(qlanth-carnall)/Ky", "Freq"}
250       ]; *)
251
252       rmsDifference = Sqrt[Total[diffs^2/Length[diffs]]];
253       labelTempate  = StringTemplate["\!\(\*SuperscriptBox[\(`ln`\),
    \(\(3\)\(+\)\)]\)"];
254       diffData    = diffs;
255       diffLabels = simplerStateLabels[[;;Length[notBad]]];
```

```
256       diffLabels = Pick[diffLabels, notBad];
257       diffPlot   = Framed[
258           ListLabelPlot[
259             diffData[[;;;;If[OddQ[numE],2,1]]],
260             diffLabels[[;;;;If[OddQ[numE],2,1]]],
261             Frame -> True,
262             PlotRange -> All,
263             ImageSize -> 1200,
264             AspectRatio -> 1/3,
265             Filling -> Axis,
266             FrameLabel -> {"",
267             "(qlanth-carnall) / \!\(\*SuperscriptBox[\(cm\), \(-1\)
    ]\)"},
268             PlotMarkers -> "OpenMarkers",
269             PlotLabel ->
270             Style[labelTempate[<|"ln" -> ln|>] <> " | " <> "\[Sigma]=
    " <>
271               ToString[Round[rmsDifference, 0.01]] <>
272               " \!\(\*SuperscriptBox[\(cm\), \(-1\)]\)\n", 20],
273             Background -> White
274           ],
275           Background -> White,
276           FrameMargins -> 50
277         ];
278       (* now place all of this in a new notebook *)
279       nb = CreateDocument[
280       {
281         TextCell[Style[
282           DisplayForm[RowBox[{SuperscriptBox[host <> ":" <> ln, "3+"
    ], "(", SuperscriptBox["f", numE], ")"}]]
283           ], "Title", TextAlignment -> Center
284         ],
285         TextCell["Energy Diagram",
286           "Section",
287           TextAlignment -> Center
288         ],
289         TextCell[energyDiagram,
290           TextAlignment -> Center
291         ],
292         TextCell["Multiplet Assignments & Energy Levels",
293           "Section",
294           TextAlignment -> Center
295         ],
296         (* TextCell[diffHistogram, TextAlignment -> Center], *)
297         TextCell[diffPlot, "Output", TextAlignment -> Center],
298         TextCell[assignmentMatches, "Output", TextAlignment -> Center
    ],
299         TextCell[diffTable, "Output", TextAlignment -> Center],
300         TextCell["Truncated Eigenstates", "Section", TextAlignment ->
     Center],
301         TextCell["These are some of the resultant eigenstates which
    add up to at least a total probability of " <> ToString[
    eigenstateTruncationProbability] <> ".", "Text", TextAlignment ->
    Center],
302         TextCell[statesTable, "Output", TextAlignment -> Center]
```

```
303       },
304       WindowSelected -> True,
305       WindowTitle -> ln <> " in " <> "LaF3" <> appToFname,
306       WindowSize -> {1600, 800}];
307       If[OptionValue["SaveData"],
308       (
309         exportFname = FileNameJoin[{workDir,OptionValue["
      OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".m"}];
310         SelectionMove[nb, After, Notebook];
311         NotebookWrite[nb, Cell["Reload Data", "Section",
      TextAlignment -> Center]];
312         NotebookWrite[nb,
313           Cell[(
314             "{rmsDifference, carnallEnergies, eigenEnergies, ln,
      carnallAssignments, simplerStateLabels, eigensys, basis,
      truncatedStates} = Import[FileNameJoin[{NotebookDirectory[],\"" <>
       StringSplit[exportFname,"/"][[-1]] <> "\"}]];"
315           ),"Input"
316         ]
317         ];
318         NotebookWrite[nb,
319           Cell[(
320             "Manipulate[First[MinimalBy[truncatedStates, Abs[First[#]
       - energy] &]], {energy,0}]"
321           ),"Input"]
322         ];
323         (* Move the cursor to the top of the notebook *)
324         SelectionMove[nb, Before, Notebook];
325         Export[exportFname,
326           {rmsDifference, carnallEnergies, eigenEnergies, ln,
      carnallAssignments, simplerStateLabels, eigensys, basis,
      truncatedStates}
327         ];
328         tinyexportFname = FileNameJoin[
329           {workDir, OptionValue["OutputDirectory"], ln <> " in " <> "
      LaF3" <> appToFname <> " - tiny.m"}
330         ];
331         tinyExport = <|"ln"->ln,
332                        "carnallEnergies"->carnallEnergies,
333                        "rmsDifference"-> rmsDifference,
334                        "eigenEnergies"-> eigenEnergies,
335                        "carnallAssignments"-> carnallAssignments,
336                        "simplerStateLabels" -> simplerStateLabels|>;
337         Export[tinyexportFname, tinyExport];
338       )
339       ];
340       If[OptionValue["NotebookSave"],
341         (
342           nbFname = FileNameJoin[{workDir,OptionValue["
      OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".nb"
      }];
343           PrintFun[">> Saving notebook to ", nbFname, " ..."];
344           NotebookSave[nb, nbFname];
345         )
346       ];
```

```
    )
    ];

    Return[{rmsDifference, carnallEnergies,
            eigenEnergies, ln,
            carnallAssignments, simplerStateLabels,
            eigensys, basis,
            truncatedStates}];
  )
];

MagneticDipoleTransitions::usage = "MagneticDipoleTransitions[numE]
    calculates the magnetic dipole transitions for the lanthanide ion
    numE in LaF3. The output is a tabular file, a raw data file, and a
     CSV file. The tabular file contains the following columns:
    \[Psi]i:simple, (* main contribution to the wavefuction |i>*)
    \[Psi]f:simple, (* main contribution to the wavefuction |j>*)
    \[Psi]i:idx,    (* index of the wavefuction |i>*)
    \[Psi]f:idx,    (* index of the wavefuction |j>*)
    Ei/K,           (* energy of the initial state in K *)
    Ef/K,           (* energy of the final state in K *)
    \[Lambda]/nm,   (* transition wavelength in nm *)
    \[CapitalDelta]\[Lambda]/nm, (* uncertainty in the transition
    wavelength in nm *)
    \[Tau]/s,       (* radiative lifetime in s *)
    AMD/s^-1        (* magnetic dipole transition rate in s^-1 *)

The raw data file contains the following keys:
    - Line Strength, (* Line strength array *)
    - AMD, (* Magnetic dipole transition rates in 1/s *)
    - fMD, (* Oscillator strengths from ground to excited states *)
    - Radiative lifetimes, (* Radiative lifetimes in s *)
    - Transition Energies / K, (* Transition energies in K *)
    - Transition Wavelengths in nm. (* Transition wavelengths in nm
    *)

The CSV file contains the same information as the tabular file.

The function also creates a notebook with a Manipulate that allows
    the user to select a wavelength interval and a lifetime power of
    ten. The results notebook is saved in the examples directory.

The function takes the following options:
    - \"Make Notebook\" -> True or False. If True, a notebook with a
    Manipulate is created. Default is True.
    - \"Print Function\" -> PrintTemporary or Print. The function
    used to print the progress of the calculation. Default is
    PrintTemporary.
    - \"Host\" -> \"LaF3\". The host material. Default is LaF3.
    - \"Wavelength Range\" -> {50,2000}. The range of wavelengths in
    nm for the Manipulate object in the created notebook. Default is
    {50,2000}.

The function returns an association containing the following keys:
    Line Strength, AMD, fMD, Radiative lifetimes, Transition Energies
```

```
    / K, Transition Wavelengths in nm.";
Options[MagneticDipoleTransitions] = {
        "Make Notebook" -> True,
        "Close Notebook" -> True,
        "Print Function" -> PrintTemporary,
        "Host" -> "LaF3",
        "Wavelength Range" -> {50,2000}};
MagneticDipoleTransitions[numE_Integer, OptionsPattern[]]:= (
  host           = OptionValue["Host"];
  \[Lambda]Range = OptionValue["Wavelength Range"];
  PrintFun       = OptionValue["Print Function"];
  {\[Lambda]min, \[Lambda]max} = OptionValue["Wavelength Range"];

  header    = {"\[Psi]i:simple","\[Psi]f:simple","\[Psi]i:idx","\[Psi
    ]f:idx","Ei/K","Ef/K","\[Lambda]/nm","\[CapitalDelta]\[Lambda]/nm"
    ,"\[Tau]/s","AMD/s^-1"};
  ln        = {"Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho","Er"
    ,"Tm","Yb"}[[numE]];
  {rmsDifference,carnallEnergies,eigenEnergies,ln,
  carnallAssignments,simplerStateLabels,eigensys,basis,
    truncatedStates} = Import["./examples/"<>ln<>" in LaF3 - example.m
    "];

  (* Some of the above are not needed here *)
  Clear[truncatedStates];
  Clear[basis];
  Clear[rmsDifference];
  Clear[carnallEnergies];
  Clear[carnallAssignments];
  If[OddQ[numE],
      eigenEnergies      = eigenEnergies[[;;;;2]];
      simplerStateLabels = simplerStateLabels[[;;;;2]];
      eigensys           = eigensys[[;;;;2]];
  ];
  eigenEnergies = eigenEnergies - eigenEnergies[[1]];

  magIon = <||>;
  PrintFun["Calculating the magnetic dipole line strength array..."];
  magIon["Line Strength"] = magIon;
  MagDipLineStrength[eigensys, numE, "Reload MagOp" -> False, "Units"
     -> "SI"];

  PrintFun["Calculating the M1 spontaneous transition rates ..."];
  magIon["AMD"] = MagDipoleRates[eigensys, numE, "Units"->"SI","
    Lifetime"->False];
  magIon["AMD"] = magIon["AMD"]/.{0.->Indeterminate};

  PrintFun["Calculating the oscillator strength for transition from
    the ground state ..."];
  magIon["fMD"]=GroundStateOscillatorStrength[eigensys, numE];

  PrintFun["Calculating the natural radiative lifetims ..."];
  magIon["Radiative lifetimes"]= 1/magIon["AMD"];

  PrintFun["Calculating the transition energies in K ..."];
```

```
transitionEnergies=Outer[Subtract,First/@eigensys,First/@eigensys];
magIon["Transition Energies / K"]=ReplaceDiagonal[
  transitionEnergies,Indeterminate];

PrintFun["Calculating the transition wavelengths in nm ..."];
magIon["Transition Wavelengths in nm"] =10^7/magIon["Transition
  Energies / K"];

PrintFun["Estimating the uncertainties in \[Lambda]/nm assuming a 1
   K uncertainty in energies."];
(*Assuming an uncertainty of 1 K in both energies used to calculate
   the wavelength*)
\[Lambda]uncertainty=Sqrt[2]*magIon["Transition Wavelengths in nm"
  ]^2*10^-7;

PrintFun["Formatting a tabular output file ..."];
numEigenvecs     = Length[eigensys];
roundedEnergies = Round[eigenEnergies, 1.];
simpleFromTo     = Outer[{#1,#2}&, simplerStateLabels,
  simplerStateLabels];
fromTo           = Outer[{#1,#2}&, Range[numEigenvecs], Range[
  numEigenvecs]];
energyPairs      = Outer[{#1,#2}&, roundedEnergies,
  roundedEnergies];
allTransitions   = {simpleFromTo,
    fromTo,
    energyPairs,
    magIon["Transition Wavelengths in nm"],
    \[Lambda]uncertainty,
    magIon["AMD"],
    magIon["Radiative lifetimes"]
};
allTransitions = (Flatten/@Transpose[Flatten[#,1]&/@allTransitions
  ]);
allTransitions = Select[allTransitions, #[[3]]!=#[[4]]&];
allTransitions = Select[allTransitions, #[[10]]>0&];
allTransitions = Transpose[allTransitions];

(*round things up*)
PrintFun["Rounding wavelengths according to estimated uncertainties
   ..."];
{roundedWaves,roundedDeltas} = Transpose[MapThread[
  RoundValueWithUncertainty,{allTransitions[[7]],allTransitions
  [[8]]}]];
allTransitions[[7]]           = roundedWaves;
allTransitions[[8]]           = roundedDeltas;

PrintFun["Rounding lifetimes and transition rates to three
  significant figures ..."];
allTransitions[[9]]           = RoundToSignificantFigures[#,3]&/@(
  allTransitions[[9]]);
allTransitions[[10]]          = RoundToSignificantFigures[#,3]&/@(
  allTransitions[[10]]);
finalTable                    = Transpose[allTransitions];
finalTable                    = Prepend[finalTable,header];
```

```
475
476 (* tabular output *)
477 basename     = ln <> " in " <> host <> " - example - " <> "MD1 -
     tabular.zip";
478 exportFname  = FileNameJoin[{"./examples",basename}];
479 PrintFun["Exporting tabular data to "<>exportFname<>" ..."];
480 exportKey    = StringReplace[basename,".zip"->".m"];
481 Export[exportFname, <|exportKey->finalTable|>];
482
483 (* raw data output *)
484 basename    = ln <> " in " <> host <> " - example - " <> "MD1 - raw
     .zip";
485 rawexportFname = FileNameJoin[{"./examples",basename}];
486 PrintFun["Exporting raw data as an association to "<>exportFname<>"
     ..."];
487 rawexportKey   = StringReplace[basename,".zip"->".m"];
488 Export[rawexportFname, <|rawexportKey->magIon|>];
489
490 (* csv output *)
491 PrintFun["Formatting and exporting a CSV output..."];
492 csvOut = Table[
493     StringJoin[Riffle[ToString[#,CForm]&/@finalTable[[i]],","]],
494 {i,1,Length[finalTable]}
495 ];
496 csvOut      = StringJoin[Riffle[csvOut, "\n"]];
497 basename    = ln <> " in " <> host <> " - example - " <> "MD1.csv";
498 exportFname = FileNameJoin[{"./examples", basename}];
499 PrintFun["Exporting csv data to "<>exportFname<>" ..."];
500 Export[exportFname, csvOut, "Text"];
501
502 If[OptionValue["Make Notebook"],
503 (
504     PrintFun["Creating a notebook with a Manipulate to select a
     wavelength interval and a lifetime power of ten ..."];
505     finalTable     = Rest[finalTable];
506     finalTable     = SortBy[finalTable,#[[7]]&];
507     opticalTable   = Select[finalTable,\[Lambda]min<=#[[7]]<=\[
     Lambda]max&];
508     pows           = Sort[DeleteDuplicates[(MantissaExponent
     [#[[9]]][[2]]-1)&/@opticalTable]];
509
510     man            = Manipulate[
511     (
512         {\[Lambda]min,\[Lambda]max} = \[Lambda]int;
513         table = Select[opticalTable,And[(\[Lambda]min<=#[[7]]<=\[
     Lambda]max),
514                   (MantissaExponent[#[[9]]][[2]]-1)==log10\[Tau
     ]]&];
515         tab   = TableForm[table,TableHeadings->{None,header}];
516         Column[{{"\[Lambda]min="<>ToString[\[Lambda]min]<>" nm","\[
     Lambda]max="<>ToString[\[Lambda]max]<>" nm",log10\[Tau]},tab}]
517     ),
518     {{\[Lambda]int,\[Lambda]Range,"\[Lambda] interval"},
519         \[Lambda]Range[[1]],
520         \[Lambda]Range[[2]],
```

```
521             50,
522             ControlType->IntervalSlider
523         },
524         {{log10\[Tau],pows[[-1]]},
525             pows
526         },
527         TrackedSymbols  :> {\[Lambda]int,log10\[Tau]},
528         SaveDefinitions -> True
529         ];
530 
531         nb = CreateDocument[{
532             TextCell[Style[DisplayForm[RowBox[{"Magnetic Dipole
    Transitions", "\n", SuperscriptBox[host<>":"<>ln,"3+"],"(",
    SuperscriptBox["f",numE],")"}]]],"Title",TextAlignment->Center],
533             (* TextCell["Magnetic Dipole Transition Lifetimes","Section
    ",TextAlignment->Center], *)
534             TextCell[man,"Output",TextAlignment->Center]
535         },
536         WindowSelected -> True,
537         WindowTitle    -> "MD1 - "<>ln<>" in "<>host,
538         WindowSize     -> {1600,800}
539         ];
540         SelectionMove[nb, After, Notebook];
541         NotebookWrite[nb, Cell["Reload Data", "Section", TextAlignment
    -> Center]];
542         NotebookWrite[nb, Cell[(
543             "magTransitions = Import[FileNameJoin[{NotebookDirectory
    [],\"" <> StringSplit[rawexportFname,"/"][[-1]] <> "\"}],\""<>
    rawexportKey<>"\"];"
544             ),"Input"]];
545         SelectionMove[nb, Before, Notebook];
546         nbFname = FileNameJoin[{workDir,"examples","MD1 - "<>ln<>" in "
    <>"LaF3"<>".nb"}];
547         PrintFun[">> Saving notebook to ",nbFname," ..."];
548         NotebookSave[nb, nbFname];
549         If[OptionValue["Close Notebook"],
550             NotebookClose[nb];
551         ];
552     )
553     ];
554 
555     Return[magIon];
556 )
```

References

[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. DOI: 10.1103/PhysRev.46.948. URL: https://link.aps.org/doi/10.1103/PhysRev.46.948.

[BS57] Hans Bethe and Edwin Salpeter. *Quantum Mechanics of One- and Two-Electron Atoms*. 1957.

[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. DOI: 10.1063/1.455853. URL: http://aip.scitation.org/doi/10.1063/1.455853 (visited on 07/02/2021).

[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.

[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.

[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. DOI: 10.1103/PhysRevB.86.125102. URL: https://link.aps.org/doi/10.1103/PhysRevB.86.125102 (visited on 07/02/2021).

[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. DOI: https://doi.org/10.1103/PhysRev.169.130.

[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. DOI: https://doi.org/10.1364/JOSAB.1.000261.

[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. DOI: 10.1103/PhysRev.127.750. URL: https://link.aps.org/doi/10.1103/PhysRev.127.750 (visited on 07/02/2021).

[Jud66] BR Judd. "Three-particle operators for equivalent electrons". In: *Physical Review* 141.1 (1966). Publisher: APS, p. 4. DOI: https://doi.org/10.1103/PhysRev.141.4.

[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.

[NK63] C. W. Nielson and George F Koster. *Spectroscopic Coefficients for the pn, dn, and fn configurations*. 1963.

[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.

[Rac43] Giulio Racah. “Theory of Complex Spectra. III”. en. In: *Physical Review* 63.9-10 (May 1943), pp. 367–382. ISSN: 0031-899X. DOI: 10.1103/PhysRev.63.367. URL: https://link.aps.org/doi/10.1103/PhysRev.63.367 (visited on 07/02/2021).

[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. DOI: https://doi.org/10.1103/PhysRev.132.280.

[TLJ99] Anne Thorne, Ulf Litzén, and Sveneric Johansson. *Spectrophysics: principles and applications*. Springer Science & Business Media, 1999.

[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. DOI: 10.1103/PhysRev.85.382. URL: https://link.aps.org/doi/10.1103/PhysRev.85.382 (visited on 01/18/2022).

[Vel00] Dobromir Velkov. “Multi-electron coefficients of fractional parentage for the p, d, and f shells”. PhD thesis. John Hopkins University, 2000.

[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.

[Wyb65] Brian G Wybourne. *Spectroscopic Properties of Rare Earths*. 1965.