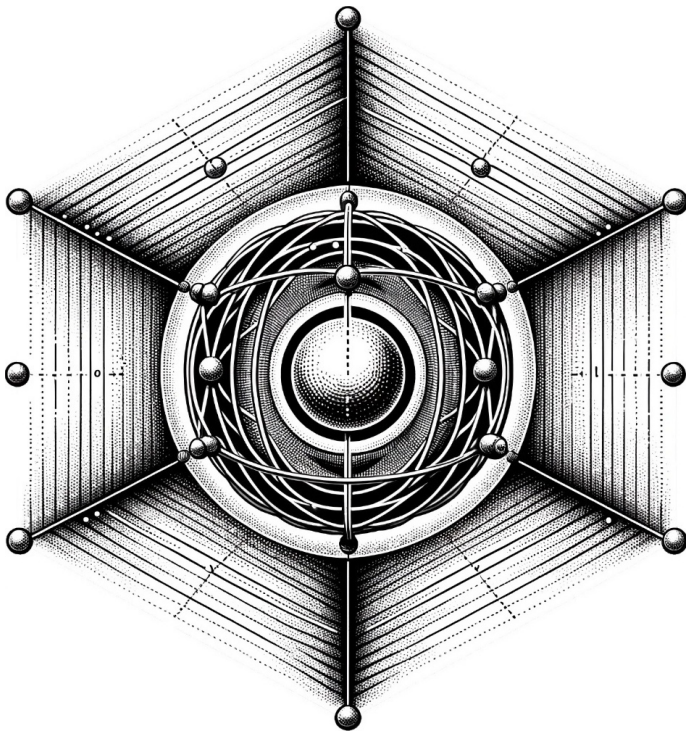

qlanth

version $|\alpha\rangle^{(6)}$

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

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 ($S\to1, P\to2, D\to3, F\to4, G\to5, H\to6, I\to7, K\to8, L\to9, M\to10, N\to11, O\to12, Q\to3, R\to14, T\to15, U\to16, V\to17$), and the final integer (if present) is the label that discriminates between several degenerate LS. This index we frequently label in the equations contained in this document with the greek letter α.

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

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

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

$\underline{\mathrm{f}}^3$ (17 LS terms)
^4S, ^4D, ^4F, ^4G, ^4I, ^2P, $^2D1$, $^2D2$, $^2F1$, $^2F2$, $^2G1$, $^2G2$, $^2H1$, $^2H2$, ^2I, ^2K, ^2L

$\underline{\mathrm{f}}^4$ (47 LS terms)
^5S, ^5D, ^5F, ^5G, ^5I, $^3P1$, $^3P2$, $^3P3$, $^3D1$, $^3D2$, $^3F1$, $^3F2$, $^3F3$, $^3F4$, $^3G1$, $^3G2$, $^3G3$, $^3H1$, $^3H2$, $^3H3$, $^3H4$, $^3I1$, $^3I2$, $^3K1$, $^3K2$, ^3L, ^3M, $^1S1$, $^1S2$, $^1D1$, $^1D2$, $^1D3$, $^1D4$, ^1F, $^1G1$, $^1G2$, $^1G3$, $^1G4$, $^1H1$, $^1H2$, $^1I1$, $^1I2$, $^1I3$, ^1K, $^1L1$, $^1L2$, ^1N

$\underline{\mathrm{f}}^5$ (73 LS terms)
^6P, ^6F, ^6H, ^4S, $^4P1$, $^4P2$, $^4D1$, $^4D2$, $^4D3$, $^4F1$, $^4F2$, $^4F3$, $^4F4$, $^4G1$, $^4G2$, $^4G3$, $^4G4$, $^4H1$, $^4H2$, $^4H3$, $^4I1$, $^4I2$, $^4I3$, $^4K1$, $^4K2$, ^4L, ^4M, $^2P1$, $^2P2$, $^2P3$, $^2P4$, $^2D1$, $^2D2$, $^2D3$, $^2D4$, $^2D5$, $^2F1$, $^2F2$, $^2F3$, $^2F4$, $^2F5$, $^2F6$, $^2F7$, $^2G1$, $^2G2$, $^2G3$, $^2G4$, $^2G5$, $^2G6$, $^2H1$, $^2H2$, $^2H3$, $^2H4$, $^2H5$, $^2H6$, $^2H7$, $^2I1$, $^2I2$, $^2I3$, $^2I4$, $^2I5$, $^2K1$, $^2K2$, $^2K3$, $^2K4$, $^2K5$, $^2L1$, $^2L2$, $^2L3$, $^2M1$, $^2M2$, ^2N, ^2O

$\underline{\mathrm{f}}^6$ (119 LS terms)
^7F, ^5S, ^5P, $^5D1$, $^5D2$, $^5D3$, $^5F1$, $^5F2$, $^5G1$, $^5G2$, $^5G3$, $^5H1$, $^5H2$, $^5I1$, $^5I2$, ^5K, ^5L, $^3P1$, $^3P2$, $^3P3$, $^3P4$, $^3P5$, $^3P6$, $^3D1$, $^3D2$, $^3D3$, $^3D4$, $^3D5$, $^3F1$, $^3F2$, $^3F3$, $^3F4$, $^3F5$, $^3F6$, $^3F7$, $^3F8$, $^3F9$, $^3G1$, $^3G2$, $^3G3$, $^3G4$, $^3G5$, $^3G6$, $^3G7$, $^3H1$, $^3H2$, $^3H3$, $^3H4$, $^3H5$, $^3H6$, $^3H7$, $^3H8$, $^3H9$, $^3I1$, $^3I2$,

$^3I3$, $^3I4$, $^3I5$, $^3I6$, $^3K1$, $^3K2$, $^3K3$, $^3K4$, $^3K5$, $^3K6$, $^3L1$, $^3L2$, $^3L3$, $^3M1$, $^3M2$, $^3M3$, ^3N, ^3O, $^1S1$, $^1S2$, $^1S3$, $^1S4$, ^1P, $^1D1$, $^1D2$, $^1D3$, $^1D4$, $^1D5$, $^1D6$, $^1F1$, $^1F2$, $^1F3$, $^1F4$, $^1G1$, $^1G2$, $^1G3$, $^1G4$, $^1G5$, $^1G6$, $^1G7$, $^1G8$, $^1H1$, $^1H2$, $^1H3$, $^1H4$, $^1I1$, $^1I2$, $^1I3$, $^1I4$, $^1I5$, $^1I6$, $^1I7$, $^1K1$, $^1K2$, $^1K3$, $^1L1$, $^1L2$, $^1L3$, $^1L4$, $^1M1$, $^1M2$, $^1N1$, $^1N2$, ^1Q

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

---

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

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

---

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

$\underline{\mathrm{f}}^9$
(73 LS terms)

---

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

$\underline{\mathrm{f}}^{10}$
(47 LS terms)

---

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

$\underline{\mathrm{f}}^{11}$
(17 LS terms)

---

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

$\underline{f}^{12}$
(7 LS terms)

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

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

^2F

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

^1S

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)

$|^3P,0,0\rangle$, $|^1S,0,0\rangle$

$(J=1)$
(3 kets)

$|^3P,1,-1\rangle$, $|^3P,1,0\rangle$, $|^3P,1,1\rangle$

$(J=2)$
(15 kets)

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

$(J = 3)$
(7 kets)

$|^3F, 3, -3\rangle$, $|^3F, 3, -2\rangle$, $|^3F, 3, -1\rangle$, $|^3F, 3, 0\rangle$, $|^3F, 3, 1\rangle$, $|^3F, 3, 2\rangle$, $|^3F, 3, 3\rangle$

$(J = 4)$
(27 kets)

$|^3F, 4, -4\rangle$, $|^3F, 4, -3\rangle$, $|^3F, 4, -2\rangle$, $|^3F, 4, -1\rangle$, $|^3F, 4, 0\rangle$, $|^3F, 4, 1\rangle$, $|^3F, 4, 2\rangle$, $|^3F, 4, 3\rangle$, $|^3F, 4, 4\rangle$, $|^3H, 4, -4\rangle$, $|^3H, 4, -3\rangle$, $|^3H, 4, -2\rangle$, $|^3H, 4, -1\rangle$, $|^3H, 4, 0\rangle$, $|^3H, 4, 1\rangle$, $|^3H, 4, 2\rangle$, $|^3H, 4, 3\rangle$, $|^3H, 4, 4\rangle$, $|^1G, 4, -4\rangle$, $|^1G, 4, -3\rangle$, $|^1G, 4, -2\rangle$, $|^1G, 4, -1\rangle$, $|^1G, 4, 0\rangle$, $|^1G, 4, 1\rangle$, $|^1G, 4, 2\rangle$, $|^1G, 4, 3\rangle$, $|^1G, 4, 4\rangle$

$(J = 5)$
(11 kets)

$|^3H, 5, -5\rangle$, $|^3H, 5, -4\rangle$, $|^3H, 5, -3\rangle$, $|^3H, 5, -2\rangle$, $|^3H, 5, -1\rangle$, $|^3H, 5, 0\rangle$, $|^3H, 5, 1\rangle$, $|^3H, 5, 2\rangle$, $|^3H, 5, 3\rangle$, $|^3H, 5, 4\rangle$, $|^3H, 5, 5\rangle$

$(J = 6)$
(26 kets)

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

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 ^3P 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 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 $\mathcal{G}_2$.

1.1.1 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.

1.1.2 $\mathcal{U}$ 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.

1.2 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

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

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 $\underline{f}^n$ 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]

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

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

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.

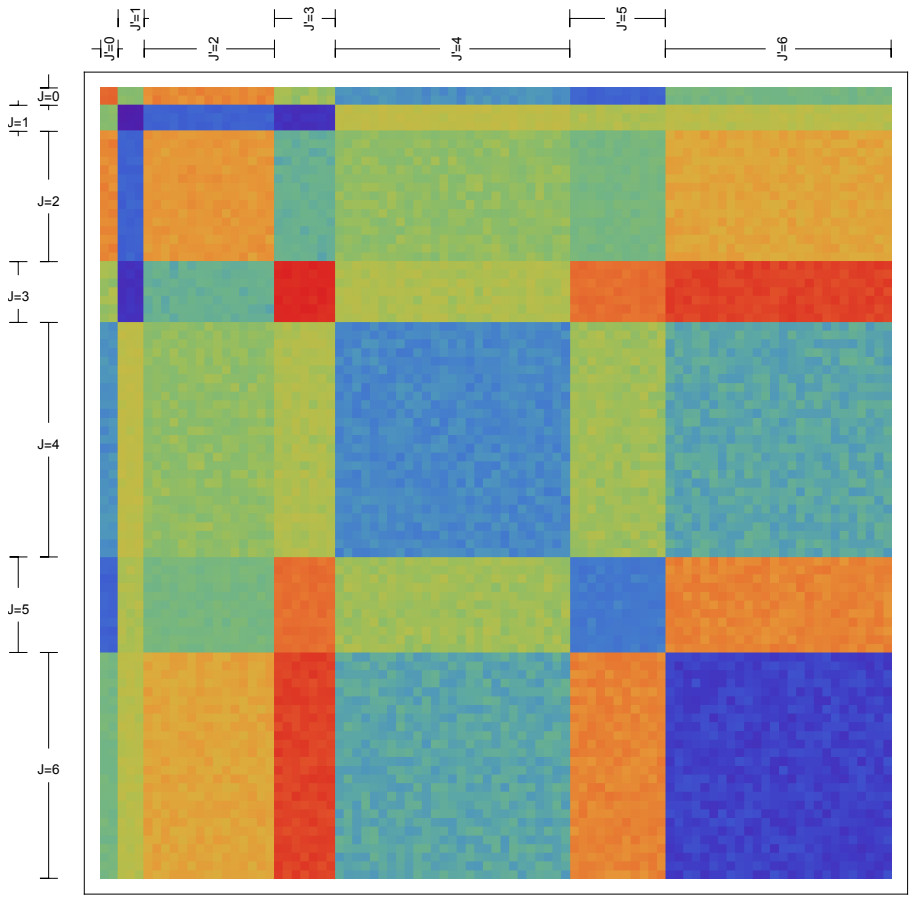

Figure 1: The block structure for $\underline{f}^2$

```
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 use 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}];)
55    ];
56
57    If[OptionValue["IncludeZeeman"],
58      (
59        PrintTemporary["Including Zeeman terms ..."];
60        {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
    ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
61        blockHam += - TeslaToKayser * (Bx * magx + By * magy + Bz *
    magz);
62      )
63    ];
64    Return[blockHam];
65  )
66 ];
```

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.

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

$$\underbrace{\hat{\mathscr{H}}_{\mathscr{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathscr{H}}_{\text{G}_2}}_{\text{G}_2 \text{ effective op}} + \underbrace{\hat{\mathscr{H}}_{\mathscr{SO}(7)}}_{\mathscr{SO}(7) \text{ effective op}} + \underbrace{\hat{\mathscr{H}}_{\curlywedge}}_{\substack{\text{effective} \\ \text{three-body}}} + \underbrace{\hat{\mathscr{H}}_{\text{cf}}}_{\text{crystal field}} + \underbrace{\hat{\mathscr{H}}_{\text{Z}}}_{\text{Zeeman}} \tag{6}$$

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

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

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

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

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

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

$$\mathscr{C}(\mathscr{G}) := \text{The Casimir operator of group } \mathscr{G}.$$

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

$$\hat{\mathscr{H}}_{\text{G}_2} = \beta\, \mathscr{C}(\mathscr{G}_2) \tag{14}$$

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

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

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

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

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 $\hat{\mathscr{H}}_{\text{k}}$: kinetic energy

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

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.

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

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]

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

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.

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

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

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

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 $\hat{\mathcal{H}}_{\text{人}}$: 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)

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

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:

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

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 the LSJ-
    reduced matrix element of the magnetic dipole operator between the
     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,
  braMJ,    ketMJ,
  matValue, magMatrix},
  (
    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' \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 use 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 f^7

In most cases all matrix elements in **qlanth** are only calculated up to and including $\underline{f}^7$. Beyond $\underline{f}^7$ adequate changes of sign are enforced to take into account the equivalence that can be made between $\underline{f}^n$ and $\underline{f}^{14-n}$ as given by &qn-4 and &qn-3.

$$\hat{\mathcal{H}} = \underbrace{\hat{\mathcal{H}}_{\mathrm{k}}}_{\text{kinetic}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:sn}}}_{\text{e:shielded nuc}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:e}}}_{\text{e:e}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:o}}}_{\text{spin-orbit}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:s}}}_{\text{spin:spin and spin:other-orbit}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:oo\oplus ecs:o}}}_{\text{spin:other-orbit ec-correlated-spin:orbit}} + \tag{71}$$

$$\underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{G_2}}}_{\text{G}_2\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(7)}}_{\mathcal{SO}(7)\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\curlywedge}}_{\text{effective three-body}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{cf}}}_{\text{crystal field}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{Z}}}_{\text{Zeeman}} \tag{72}$$

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):

$$\zeta, T^{(2)}, T^{(3)}, T^{(4)}, T^{(6)}, T^{(7)}, T^{(8)}, B_q^{(k)} \tag{73}$$

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 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])

$$\hat{\mathcal{S}}(\psi,\psi') \coloneqq |\langle\psi|\hat{\mu}|\psi'\rangle|^2 = |\langle\psi|\hat{\mu}_x|\psi'\rangle|^2 + |\langle\psi|\hat{\mu}_y|\psi'\rangle|^2 + |\langle\psi|\hat{\mu}_z|\psi'\rangle|^2 \tag{74}$$

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

$$\mathrm{A_{MD}}(|\psi_i\rangle \to |\psi_f\rangle) = \frac{16\pi^3\mu_0}{3h}\frac{n^3}{\lambda^3}\frac{\hat{\mathcal{S}}(\psi_i,\psi_f)}{g_i}, \tag{75}$$

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]

$$\mathrm{f_{MD}}\left(|\psi_g\rangle \to |\psi_e\rangle\right) = \frac{8\pi^2 m_e}{3\,hc\,e^2}\frac{n}{\lambda}\frac{\hat{S}(\psi_g,\psi_e)}{g_g} \tag{76}$$

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, $g_g = 1$.

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

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 $\mathcal{F}$ig-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\"
```

```
    .
8
9   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.
10
11  The list startValues is a list with all of the parameters needed to
        define the Hamiltonian (including the initial values for
        problemVars).
12
13  The function saves the solution to a file. The file is named with a
        prefix (controlled by the option \"FilePrefix\") and a UUID. The
        file is saved in the log sub-directory as a .m file.
14
15  Here's a description of the different parts of this function: first
        the Hamiltonian is assembled and simplified using the given
        simplifications. Then the intermediate coupling basis is
        calculated using the free-ion parameters for the given lanthanide.
         The Hamiltonian is then changed to the intermediate coupling
        basis and truncated. The truncated Hamiltonian is then compiled
        into a function that can be used to calculate the energy levels of
         the truncated Hamiltonian. The function that calculates the
        energy levels is then used to fit the experimental data. The
        fitting is done using FindMinimum with the Levenberg-Marquardt
        method.
16
17  The function returns an association with the following keys:
18
19  \"bestRMS\" which is the best \[Sigma] value found.
20  \"bestParams\" which is the best set of parameters found.
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.
70    \"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.
71  ";
72  Options[ClassicalFit] = {
73    "MaxHistory"       -> 200,
74    "MaxIterations"    -> 100,
75    "FilePrefix"       -> "calcs",
76    "ProgressView"     -> True,
77    "TruncationEnergy" -> Automatic,
78    "AccuracyGoal"     -> 5,
79    "PrintFun"         -> PrintTemporary,
80    "SlackChannel"     -> None,
81    "ProgressView"     -> True,
82    "SignatureCheck"   -> False,
83    "AddConstantShift" -> False,
84    "SaveEigenvectors" -> False,
85    "AppendToLogFile"  -> <||>,
86    "MagneticSimplifier" -> {
87      M2 -> 56/100 M0,
88      M4 -> 31/100 M0,
89      P4 -> 1/2 P2,
90      P6 -> 1/10 P2
91      },
92    "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}, *)
Module[{},
(
  solCompendium   = <||>;
  addShift       = OptionValue["AddConstantShift"];
  ln             = theLanthanides[[numE]];
  maxHistory     = OptionValue["MaxHistory"];
  maxIterations  = OptionValue["MaxIterations"];
  logFilePrefix  = If[OptionValue["FilePrefix"] == "",
                     ToString[theLanthanides[[numE]]],
                     OptionValue["FilePrefix"]
                   ];
  accuracyGoal  = OptionValue["AccuracyGoal"];
  slackChan     = OptionValue["SlackChannel"];
  PrintFun      = OptionValue["PrintFun"];
  freeIonSymbols       = OptionValue["FreeIonSymbols"];
  runningInteractive   = (Head[$ParentLink] === LinkObject);
  magneticSimplifier   = OptionValue["MagneticSimplifier"];
  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 *)
334       PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
335       (* intermediateHam            = Transpose[basisChanger].ham.
    basisChanger; *)
336       (* Return[{basisChangerBlocks, ham}]; *)
337       intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks];
338       PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
339       Do[
340         intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
341         {rowIdx, 1, numBlocks},
342         {colIdx, 1, numBlocks}
343       ];
344       intermediateHam = BlockMatrixMultiply[BlockTranspose[
    basisChangerBlocks],intermediateHam];
345       PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
346       Do[
347         intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
348         {rowIdx, 1, numBlocks},
349         {colIdx, 1, numBlocks}
350       ];
351       (* using the truncation indices truncate that one *)
352       PrintFun["Truncating the Hamiltonian ..."];
```

```
353       truncatedIntermediateHam = TruncateBlockArray[intermediateHam,
      truncationIndices, blockShifts];
354       (* these are the basis vectors for the truncated hamiltonian *)
355       PrintFun["Saving the truncated intermediate basis ..."];
356       truncatedIntermediateBasis = basisChanger[[All,
      truncationIndices]];
357
358       PrintFun["Compiling a function for the truncated Hamiltonian
      ..."];
359       (* compile a function that will calculate the truncated
      Hamiltonian given the parameters in allVars, this is the function
      to be use in fitting *)
360       compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
      Evaluate[truncatedIntermediateHam]];
361       (* save the compiled function *)
362       PrintFun["Saving the compiled function for the truncated
      Hamiltonian and the truncated intermediate basis ..."];
363       Export[compiledIntermediateFname, {
      compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
364      )
365      ];
366
367      truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
368      PrintFun["The truncated Hamiltonian has a dimension of ",
      truncationUmbral, "x", truncationUmbral, " ..."];
369      presentDataIndices = Select[presentDataIndices, # <=
      truncationUmbral &];
370      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
```

```
...];
eigenEnergies = eigenEnergies - eigenEnergies[[1]];
PrintFun["Calculating the linear approximant to each eigenvalue
...];
allVarsVec = Transpose[{allVars}];
p0 = Transpose[{fullSolVec}];
linMat = {};
If[addShift,
  tail = -2,
  tail = -1];
Do[
  (
    aVarPosition = Position[allVars, aVar][[1, 1]];
    isolationValues = ConstantArray[0, Length[allVars]];
    isolationValues[[aVarPosition]] = 1;
    dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
constraints]];
    Do[
      isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
dVar[[2]],
      {dVar, dependentVars}
    ];
    perHam  = compileIntermediateTruncatedHam @@ isolationValues;
    lin     = FirstOrderPerturbation[Last /@ states, perHam];
    linMat  = Append[linMat, lin];
  ),
  {aVar, constrainedProblemVarsList[[;;tail]]}
];
PrintFun["Removing the gradient of the ground state ..."];
linMat = (# - #[[1]] & /@ linMat);
PrintFun["Transposing derivative matrices into columns ..."];
linMat  = Transpose[linMat];

PrintFun["Calculating the eigenvalue vector at solution ..."];
\[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
PrintFun["Putting together the experimental vector ..."];
\[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
]]}];
problemVarsVec = If[addShift,
  Transpose[{constrainedProblemVarsList[[;;-2]]}],
  Transpose[{constrainedProblemVarsList}]
];
indepSolVecVec = Transpose[{indepSolVec}];
PrintFun["Calculating the difference between eigenvalues at
solution ..."];
diff = If[linMat=={},
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
presentDataIndices]].(problemVarsVec - indepSolVecVec)
];
PrintFun["Calculating the sum of squares of differences around
solution ... "];
sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
PrintFun["Calculating the minimum (which should coincide with sol
 ..."];
```

```
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=={},
551       {{0}},
552       \[Sigma]exp^2 * Inverse[hess]
553     ];
554     bestRMS = Sqrt[minpoly / degressOfFreedom];
555     solCompendium["truncatedDim"] = truncationUmbral;
556     solCompendium["fittedLevels"] = Length[presentDataIndices];
557     solCompendium["actualSteps"] = steps;
558     solCompendium["bestRMS"]     = bestRMS;
559     solCompendium["solWithUncertainty"] = solWithUncertainty;
560     solCompendium["problemVars"] = problemVars;
561     solCompendium["paramSols"]   = paramSols;
562     solCompendium["rmsHistory"]  = rmsHistory;
563     solCompendium["Appendix"]    = OptionValue["AppendToLogFile"];
564     solCompendium["timeTaken/s"] = timeTaken;
565     solCompendium["bestParams"]  = sol[[2]];
566     If[OptionValue["SaveEigenvectors"],
567         solCompendium["states"] = {#[[1]] + \[Epsilon]Best, #[[2]]}
        &/@ (Chop /@ ShiftedLevels[states]),
568         (
569           finalEnergies = Sort[First /@ states];
570           finalEnergies = finalEnergies - finalEnergies[[1]];
571           finalEnergies = finalEnergies + \[Epsilon]Best;
572           finalEnergies  = Chop /@ finalEnergies;
573           solCompendium["energies"] = finalEnergies;
574       )
575     ];
576     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];
  ];
);
```

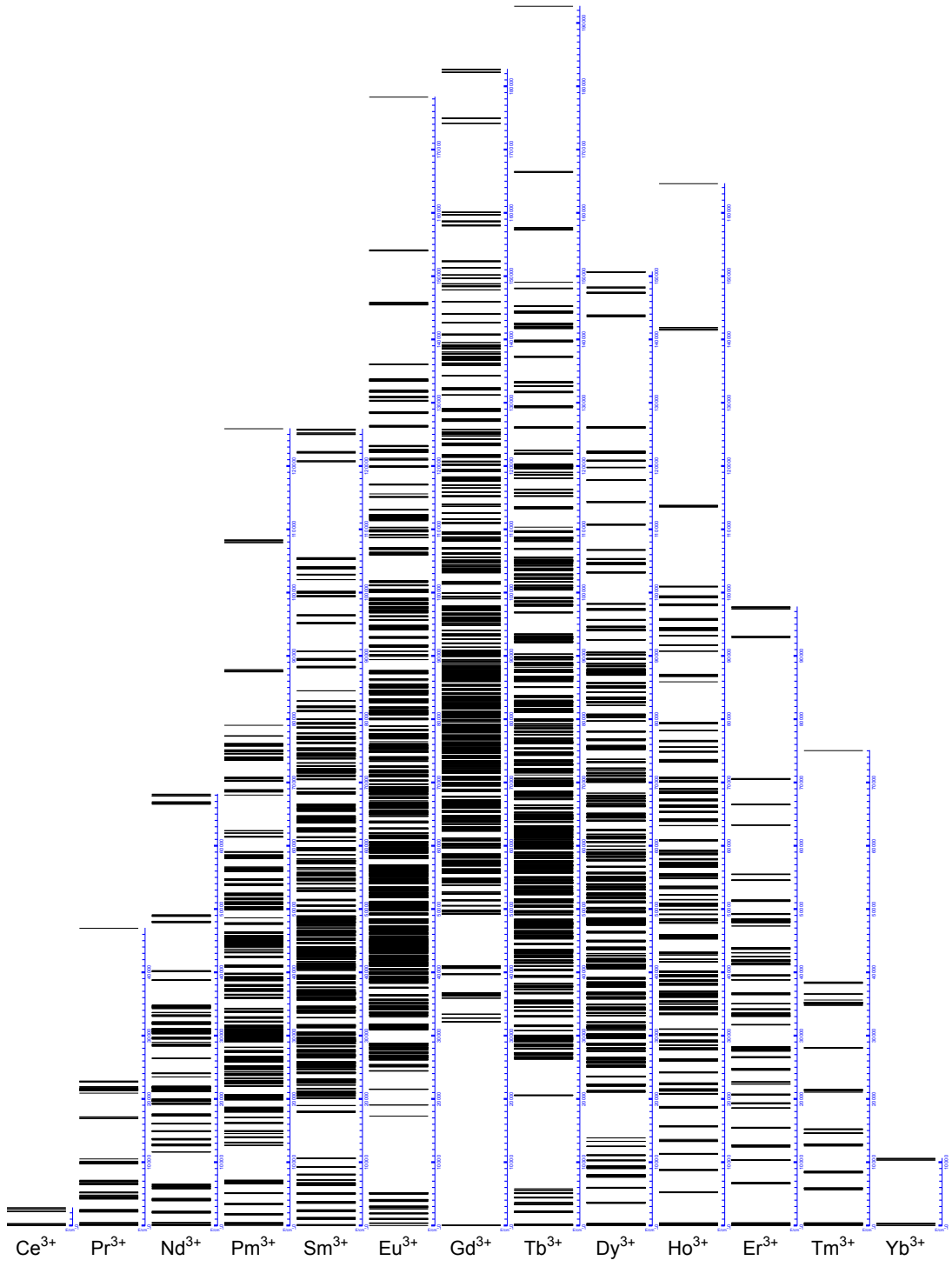

Figure 3: Dieke plot for lanthanum fluoride from data generated by **qlanth**.

```
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},
  (
    source = OptionValue["Source"];
    params = Which[source=="Carnall",
             (Association[Carnall["data"][Ln]])
             ];
    (*If a free ion then all the parameters from the crystal field
    are set to zero*)
    If[OptionValue["Free Ion"],
      Do[params[cfSymbol] = 0, {cfSymbol, cfSymbols}]
    ];
    params[F0] = 0;
    params[M2] = 0.56*params[M0]; (*See Carnall 1989,Table I,caption,
    probably fixed based on HF values*)
    params[M4] = 0.31*params[M0]; (*See Carnall 1989,Table I,caption,
    probably fixed based on HF values*)
    params[P0] = 0;
    params[P4] = 0.5*params[P2];  (*See Carnall 1989,Table I,caption,
    probably fixed based on HF values*)
    params[P6] = 0.1*params[P2];  (*See Carnall 1989,Table I,caption,
    probably fixed based on HF values*)
    params[gs] = OptionValue["gs"];
    {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[F0
    ], params[F2], params[F4], params[F6]];
    params[E0] = 0;
    If[
      Not[OptionValue["With Uncertainties"]],
      Return[params],
      (
        uncertain     = Association[Carnall["annotations"][Ln]];
        uncertainKeys = Keys[uncertain];
        uncertain     = If[# == "Not allowed to vary in fitting." ||
    # == "Interpolated",
            0., #] & /@ uncertain;
        paramKeys = Keys[params];
        uncertainVals = Sort[Intersection[paramKeys, uncertainKeys]]
    /. Association[uncertain];
        uncertainRules = MapThread[Rule, {Sort[uncertainKeys],
    uncertainVals}];
```

```
        Which[
          MemberQ[{"Ce", "Yb"}, Ln],
          (
            subsetL = {F0};
            subsetR = {0};
          ),
          True,
          (
          subsetL = {F0, M2, M4, P0, P4, P6, E0, E1, E2, E3};
          subsetR = {0, M0*0.65, M0*0.31, 0, P2*0.5, P2*0.1,
            0,
            Sqrt[(196 F2^2)/164025 + (49 F4^2)/88209 + (122500 F6^2)
    /134165889],
            Sqrt[F2^2/4100625 + F4^2/10673289 + (30625 F6^2)
    /2743558264161],
            Sqrt[F2^2/18225 + (4 F4^2)/1185921 + (30625 F6^2)
    /1803785841]};
          )
        ];
        uncertainRules = Join[uncertainRules, MapThread[Rule, {
    subsetL,subsetR /. uncertainRules}]];
        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-73 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 *)
\[Mu]0  = 4 \[Pi]*10^-7;             (* magnetic permeability in
    vacuum in SI *)
hPlanck = 6.62607015*10^-34;         (* Planck's constant in SI *)
\[Mu]B  = 9.2740100783*10^-24;       (* Bohr magneton in SI *)
me      = 9.1093837015*10^-31;       (* electron mass in SI *)
cLight  = 2.99792458*10^8;           (* speed of light in SI *)
eCharge = 1.602176634*10^-19;        (* elementary charge in SI *)
αFine = 1/137.036;             (* fine structure constant in SI *)
bohrRadius   = 5.29177210903*10^-11; (* Bohr radius in SI *)

(* 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 *)
eVtoKayser    = 8065.54429;   (* 1 eV = 8065.54429 cm^-1 *)
KaysertoeV    = 1/eVtoKayser; (* 1 cm^-1 = 1/8065.54429 eV *)
TeslaToKayser = 2 * \[Mu]B / hPlanck / cLight / 100;

EndPackage[];
```

10 Notation

orbital angular momentum operator of a single electron

$$\overbrace{\hat{l}} \tag{77}$$

total orbital angular momentum operator

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

spin angular momentum operator of a single electron

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

total spin angular momentum operator

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

Shorthand for all other quantum numbers

$$\overbrace{\Lambda} \tag{81}$$

orbital angular momentum number

$$\overbrace{\underline{\ell}} \tag{82}$$

spinning angular momentum number

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

Coulomb non-central potential

$$\overbrace{\hat{\mathfrak{C}}} \tag{84}$$

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

$$\overbrace{\langle \Lambda LS \| \hat{O} \| \Lambda' L' S' \rangle} \tag{85}$$

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

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

Spectroscopic term αLS in Russel-Saunders notation

$$\overbrace{{}^{2S+1}\alpha L \equiv |\alpha LS\rangle} \tag{87}$$

spherical tensor operator of rank k

$$\overbrace{\hat{X}^{(k)}} \tag{88}$$

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

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

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

$$\overbrace{\left(\underline{\ell}^{n-1} \alpha' L' S' \middle\} \underline{\ell}^n \alpha LS\right)} \tag{90}$$

11 Definitions

$$\overbrace{[\![x]\!]}^{\text{two plus one}} \coloneqq 2x+1 \tag{91}$$

$$\overbrace{\hat{u}^{(k)}}^{\text{irreducible unit tensor operator of rank k}} \tag{92}$$

$$\overbrace{\hat{U}^{(k)} \coloneqq \sum_{i=1}^{n} \hat{u}^{(k)}}^{\text{symmetric unit tensor operator for n equivalent electrons}} \tag{93}$$

$$\overbrace{\mathcal{C}_q^{(k)} \coloneqq \sqrt{\frac{4\pi}{2k+1}}\mathrm{Y}_q^{(k)}}^{\text{Renormalized spherical harmonics}} \tag{94}$$

$$\overbrace{\triangle(j_1,j_2,j_3) \coloneqq \begin{cases} 1 & \text{if } j_1 = (j_2+j_3),(j_2+j_3-1),\ldots,|j_2-j_3| \\ 0 & \text{otherwise} \end{cases}}^{\text{Triangle ``delta'' between } j_1,j_2,j_3} \tag{95}$$

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 H Shortley. The Theory of Atomic Spectra, 1935.

+ 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.

```

```
+ 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.

+ 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.

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

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

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

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

+ 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.

+ 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.


---------------------------------------------------------------- *)
```

```
105
106 BeginPackage["qlanth`"];
107 Needs["qonstants`"];
108 Needs["qplotter`"];
109 Needs["misc`"];
110
111 paramAtlas = "
112 E0: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
113 E1: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
114 E2: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
115 E3: linear combination of F_k, see eqn. (2-80) in Wybourne 1965
116
117 ζ: spin-orbit strength parameter.
118
119 F0: Direct Slater integral F^0, produces an overall shift of all
        energy levels.
120 F2: Direct Slater integral F^2
121 F4: Direct Slater integral F^4, possibly constrained by ratio to F^2
122 F6: Direct Slater integral F^6, possibly constrained by ratio to F^2
123
124 M0: 0th Marvin integral
125 M2: 2nd Marvin integral
126 M4: 4th Marvin integral
127 \[Sigma]SS: spin-spin override, if 0 spin-spin is omitted, if 1 then
        spin-spin is included
128
129 T2:  three-body effective operator parameter T^2 (non-orthogonal)
130 T2p: three-body effective operator parameter T^2' (orthogonalized T2)
131 T3:  three-body effective operator parameter T^3
132 T4:  three-body effective operator parameter T^4
133 T6:  three-body effective operator parameter T^6
134 T7:  three-body effective operator parameter T^7
135 T8:  three-body effective operator parameter T^8
136
137 T11:  three-body effective operator parameter T^11
138 T11p: three-body effective operator parameter T^11'
139 T12:  three-body effective operator parameter T^12
140 T14:  three-body effective operator parameter T^14
141 T15:  three-body effective operator parameter T^15
142 T16:  three-body effective operator parameter T^16
143 T17:  three-body effective operator parameter T^17
144 T18:  three-body effective operator parameter T^18
145 T19:  three-body effective operator parameter T^19
146
147 P0: pseudo-magnetic parameter P^0
148 P2: pseudo-magnetic parameter P^2
149 P4: pseudo-magnetic parameter P^4
150 P6: pseudo-magnetic parameter P^6
151
152 gs: electronic gyromagnetic ratio
153
154 α: Trees' parameter α describing configuration interaction via the
        Casimir operator of SO(3)
155 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)
";
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;

```

```
252 CFPTable;
253 CFPTerms;
254 Carnall;
255 CasimirG2;
256 CasimirSO3;
257 CasimirSO7;
258
259 Cqk;
260 CrystalField;
261 Dk;
262 ElectrostaticConfigInteraction;
263 Electrostatic;
264
265 ElectrostaticTable;
266 EnergyLevelDiagram;
267 EnergyStates;
268 ExportMZip;
269 BasisTableGenerator;
270 EigenLever;
271 EtoF;
272 ExportmZip;
273 fsubk;
274 fsupk;
275
276 FindNKLSTerm;
277 FindSL;
278
279 FreeHam;
280 FtoE;
281 GG2U;
282 GSO7W;
283 GenerateCFP;
284 GenerateCFPAssoc;
285
286 GenerateCFPTable;
287 GenerateCrystalFieldTable;
288 GenerateElectrostaticTable;
289 GenerateReducedUkTable;
290 GenerateReducedV1kTable;
291
292 GenerateSOOandECSOLSTable;
293 GenerateSOOandECSOTable;
294 GenerateSpinOrbitTable;
295 GenerateSpinSpinTable;
296 GenerateT22Table;
297
298 GenerateThreeBodyTables;
299 GenerateThreeBodyTables;
300 Generator;
301 GroundStateOscillatorStrength;
302 HamMatrixAssembly;
303 HamiltonianForm;
304
305 HamiltonianMatrixPlot;
306 HoleElectronConjugation;
```

```
IntermediateSolver;
IonSolver;
ImportMZip;
JJBlockMatrix;
JJBlockMagDip;
JJBlockMatrixFileName;

JJBlockMatrixTable;
LabeledGrid;
ListRepeater;
LoadAll;
LoadCFP;
LoadCarnall;

LoadChenDeltas;
LoadElectrostatic;
LoadGuillotParameters;
LoadParameters;
LoadSOOandECSO;

LoadSOOandECSOLS;
LoadSpinOrbit;
LoadSpinSpin;
LoadSymbolicHamiltonians;
LoadT11;

LoadT22;
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);
```

```
415
416 (* ############################################################ *)
417 (* ########################## MISC ############################ *)
418
419 TPO::usage = "TPO[x, y, ...] gives the product of 2x+1, 2y+1, ...";
420 TPO[args__] := Times @@ ((2*# + 1) & /@ {args});
421
422 Phaser::usage = "Phaser[x] gives (-1)^x.";
423 Phaser[exponent_] := ((-1)^exponent);
424
425 TriangleCondition::usage = "TriangleCondition[a, b, c] evaluates
      the triangle condition on a, b, and c.";
426 TriangleCondition[a_, b_, c_] := (Abs[b - c] <= a <= (b + c));
427
428 TriangleAndSumCondition::usage = "TriangleAndSumCondition[a, b, c]
      evaluates the joint satisfaction of the triangle and sum
      conditions.";
429 TriangleAndSumCondition[a_, b_, c_] := (
430   And[
431     Abs[b - c] <= a <= (b + c),
432     IntegerQ[a + b + c]
433     ]
434   );
435
436 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.
      ";
437 SquarePrimeToNormal[squarePrime_] :=
438 (
439   radical = Product[Prime[idx1 - 1] ^ Part[squarePrime, idx1], {
      idx1, 2, Length[squarePrime]}];
440   radical = radical /. {"A" -> 10, "B" -> 11, "C" -> 12, "D" ->
      13};
441   val = squarePrime[[1]] * Sqrt[radical];
442   Return[val];
443 );
444
445 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.
446 The function additionally takes an option \"Print\" that if set to
      True, will print the symbols that were not present in the given
      association.";
447 Options[ParamPad] = {"Print" -> True}
448 ParamPad[params_, OptionsPattern[]] := (
449   notPresentSymbols = Complement[paramSymbols, Keys[params]];
450   If[OptionValue["Print"],
```

```
451       Print["Following symbols were not given and are being set to 0:
      ",
452       notPresentSymbols]
453     ];
454     newParams = Transpose[{paramSymbols, ConstantArray[0, Length[
      paramSymbols]]}];
455     newParams = (#[[1]] -> #[[2]]) & /@ newParams;
456     newParams = Association[newParams];
457     newParams = Join[newParams, params];
458     Return[newParams];
459     )
460
461   (* ############################################################ *)
462   (* ###################### Racah Algebra ###################### *)
463
464   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.";
465   ReducedUk[numE_, l_, SL_, SpLp_, k_]:=Module[
466     {spin, orbital, Uk, S, L, Sp, Lp, Sb, Lb, parentSL, cfpSL,
      cfpSpLp, Ukval, SLparents, SLpparents, commonParents, phase},
467       {spin, orbital} = {1/2, 3};
468       {S, L}          = FindSL[SL];
469       {Sp, Lp}        = FindSL[SpLp];
470       If[Not[S == Sp],
471         Return[0]
472       ];
473       cfpSL       = CFP[{numE, SL}];
474       cfpSpLp     = CFP[{numE, SpLp}];
475       SLparents   = First /@ Rest[cfpSL];
476       SLpparents  = First /@ Rest[cfpSpLp];
477       commonParents = Intersection[SLparents, SLpparents];
478       Uk = Sum[(
479         {Sb, Lb} = FindSL[\[Psi]b];
480         Phaser[Lb] *
481           CFPAssoc[{numE, SL, \[Psi]b}] *
482           CFPAssoc[{numE, SpLp, \[Psi]b}] *
483           SixJay[{orbital, k, orbital}, {L, Lb, Lp}]
484       ),
485       {\[Psi]b, commonParents}
486       ];
487       phase      = Phaser[orbital + L + k];
488       prefactor = numE * phase * Sqrt[TPO[L,Lp]];
489       Ukval      = prefactor*Uk;
490       Return[Ukval];
491   ]
492
493   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'.";
494   Ck[orbital_, k_] := (-1)^orbital * TPO[orbital] * ThreeJay[{orbital
      , 0}, {k, 0}, {orbital, 0}];
495
496   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.";
497 SixJay[{j1_, j2_, j3_}, {j4_, j5_, j6_}] := (
498   sixJayval = Which[
499     Not[TriangleAndSumCondition[j1, j2, j3]],
500     0,
501     Not[TriangleAndSumCondition[j1, j5, j6]],
502     0,
503     Not[TriangleAndSumCondition[j4, j2, j6]],
504     0,
505     Not[TriangleAndSumCondition[j4, j5, j3]],
506     0,
507     True,
508     SixJSymbol[{j1, j2, j3}, {j4, j5, j6}]];
509   SixJay[{j1, j2, j3}, {j4, j5, j6}] = sixJayval);
510
511 ThreeJay::usage = "ThreeJay[{j1, m1}, {j2, m2}, {j3, m3}] gives the
     value of the Wigner 3j-symbol and memorizes the computed value.";
512 ThreeJay[{j1_, m1_}, {j2_, m2_}, {j3_, m3_}] := (
513  threejval = Which[
514    Not[(m1 + m2 + m3) == 0],
515    0,
516    Not[TriangleCondition[j1,j2,j3]],
517    0,
518    True,
519    ThreeJSymbol[{j1, m1}, {j2, m2}, {j3, m3}]
520    ];
521  ThreeJay[{j1, m1}, {j2, m2}, {j3, m3}] = threejval);
522
523 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.";
524 ReducedV1k[numE_, SL_, SpLp_, k_]:=Module[
525   {Vk1, S, L, Sp, Lp, Sb, Lb, spin, orbital, cfpSL, cfpSpLp,
526   SLparents, SpLpparents, commonParents, prefactor},
527   (
528     {spin, orbital} = {1/2, 3};
529     {S, L}          = FindSL[SL];
530     {Sp, Lp}        = FindSL[SpLp];
531     cfpSL           = CFP[{numE, SL}];
532     cfpSpLp         = CFP[{numE, SpLp}];
533     SLparents       = First /@ Rest[cfpSL];
534     SpLpparents     = First /@ Rest[cfpSpLp];
535     commonParents = Intersection[SLparents, SpLpparents];
536     Vk1 = Sum[(
537         {Sb, Lb} = FindSL[\[Psi]b];
538         Phaser[(Sb + Lb + S + L + orbital + k - spin)] *
539         CFPAssoc[{numE, SL, \[Psi]b}] *
540         CFPAssoc[{numE, SpLp, \[Psi]b}] *
541         SixJay[{S, Sp, 1}, {spin, spin, Sb}] *
542         SixJay[{L, Lp, k}, {orbital, orbital, Lb}]
543       ),
544     {\[Psi]b, commonParents}
545     ];
```

```
546       prefactor = numE * Sqrt[spin * (spin + 1) * TPO[spin, S, L, Sp,
      Lp] ];
547       Return[prefactor * Vk1];
548     )
549   ];
550
551   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.";
552   Options[GenerateReducedUkTable] = {"Export" -> True, "Progress" ->
      True};
553   GenerateReducedUkTable[numEmax_Integer:7, OptionsPattern[]]:= (
554     numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
      AllowedNKSLTerms[#]]&/@Range[1, numEmax]] * 4;
555     Print["Calculating " <> ToString[numValues] <> " values for Uk k
      =0,2,4,6."];
556     counter = 1;
557     If[And[OptionValue["Progress"], frontEndAvailable],
558     progBar = PrintTemporary[
559         Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
560           counter}]]]
561       ];
562     ReducedUkTable = Table[
563       (
564         counter = counter+1;
565         {numE, 3, SL, SpLp, k} -> SimplifyFun[ReducedUk[numE, 3, SL,
      SpLp, k]]
566       ),
567       {numE, 1, numEmax},
568       {SL,   AllowedNKSLTerms[numE]},
569       {SpLp, AllowedNKSLTerms[numE]},
570       {k, {0, 2, 4, 6}}
571     ];
572     ReducedUkTable = Association[Flatten[ReducedUkTable]];
573     ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedUkTable.m"}];
574     If[And[OptionValue["Progress"], frontEndAvailable],
575       NotebookDelete[progBar]
576     ];
577     If[OptionValue["Export"],
578       (
579         Print["Exporting to file " <> ToString[ReducedUkTableFname]];
580         Export[ReducedUkTableFname, ReducedUkTable];
581       )
582     ];
583     Return[ReducedUkTable];
584   )
585
586   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"}];
  If[OptionValue["Export"],
    (
      Print["Exporting to file "<>ToString[exportFname]];
      Export[exportFname, ReducedV1kTable];
    )
  ];
  Return[ReducedV1kTable];
)

(* ###################### Racah Algebra ###################### *)
(* ########################################################### *)

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

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;
```

```
635       termsWithSameSpin = StringCases[terms, ToString[
    spinMultiplicity] ~~ __];
636       termsWithSameSpin = Flatten[termsWithSameSpin];
637       If[Not[{S, L} == {Sp, Lp}],
638         Return[0]
639       ];
640       prefactor = 1/2 * Abs[Ck[orbital, k]]^2;
641       summand1 = Sum[(
642           ReducedUkTable[{numE, orbital, SL, NKSL,  k}] *
643           ReducedUkTable[{numE, orbital, SL, NKSLp, k}]
644           ),
645         {SL, termsWithSameSpin}
646       ];
647       summand1 = 1 / TPO[L] * summand1;
648       summand2 = (
649         KroneckerDelta[NKSL, NKSLp] *
650           (numE *(4*orbital + 2 - numE)) /
651           ((2*orbital + 1) * (4*orbital + 1))
652         );
653       fsubkVal = prefactor*(summand1 - summand2);
654       Return[fsubkVal];
655     )
656   ];
657
658   fsupk::usage = "fsupk[numE, orbital, SL, SLp, k] gives the
      superscripted Slater integral f^k = Subscript[f, k] * Subscript[D,
       k].";
659   fsupk[numE_, orbital_, NKSL_, NKSLp_ ,k_]:= (
660     Dk[k] * fsubk[numE, orbital, NKSL, NKSLp, k]
661     )
662
663   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.";
664   Dk[k_] := {1, 225, 1089, 184041/25}[[k/2+1]];
665
666   FtoE::usage = "FtoE[F0, F2, F4, F6] calculates the Racah parameters
       {E0, E1, E2, E3} corresponding to the given Slater integrals.
667   See eqn. 2-80 in Wybourne.
668   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.";
669   FtoE[F0_, F2_, F4_, F6_]:=Module[
670     {E0, E1, E2, E3},
671     (
672       E0 = (F0 - 10*F2/Dk[2] - 33*F4/Dk[4] - 286*F6/Dk[6]);
673       E1 = (70*F2/Dk[2] + 231*F4/Dk[4] + 2002*F6/Dk[6])/9;
674       E2 = (F2/Dk[2] - 3*F4/Dk[4] + 7*F6/Dk[6])/9;
675       E3 = (5*F2/Dk[2] + 6*F4/Dk[4] - 91*F6/Dk[6])/3;
676       Return[{E0, E1, E2, E3}];
677     )
678   ];
679
```

```
680  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.";
681  EtoF[E0_, E1_, E2_, E3_]:=Module[
682    {F0, F2, F4, F6},
683    (
684      F0 = 1/7      (7 E0 + 9 E1);
685      F2 = 75/14    (E1 + 143 E2 + 11 E3);
686      F4 = 99/7     (E1 - 130 E2 + 4 E3);
687      F6 = 5577/350 (E1 + 35 E2 - 7 E3);
688      Return[{F0, F2, F4, F6}];
689    )
690  ];
691
692  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\".";
693  Options[Electrostatic] = {"Coefficients" -> "Slater"};
694  Electrostatic[{numE_, NKSL_, NKSLp_}, OptionsPattern[]]:=Module[
695    {fsub0, fsub2, fsub4, fsub6,
696     esub0, esub1, esub2, esub3,
697     fsup0, fsup2, fsup4, fsup6,
698     eMatrixVal, orbital},
699    (
700      orbital = 3;
701      Which[
702        OptionValue["Coefficients"] == "Slater",
703        (
704          fsub0 = fsubk[numE, orbital, NKSL, NKSLp, 0];
705          fsub2 = fsubk[numE, orbital, NKSL, NKSLp, 2];
706          fsub4 = fsubk[numE, orbital, NKSL, NKSLp, 4];
707          fsub6 = fsubk[numE, orbital, NKSL, NKSLp, 6];
708          eMatrixVal = fsub0*F0 + fsub2*F2 + fsub4*F4 + fsub6*F6;
709        ),
710        OptionValue["Coefficients"] == "Racah",
711        (
712          fsup0 = fsupk[numE, orbital, NKSL, NKSLp, 0];
713          fsup2 = fsupk[numE, orbital, NKSL, NKSLp, 2];
714          fsup4 = fsupk[numE, orbital, NKSL, NKSLp, 4];
715          fsup6 = fsupk[numE, orbital, NKSL, NKSLp, 6];
716          esub0 = fsup0;
717          esub1 = 9/7*fsup0 +   1/42*fsup2 +   1/77*fsup4 +  1/462*
       fsup6;
718          esub2 =             143/42*fsup2 - 130/77*fsup4 + 35/462*
       fsup6;
719          esub3 =              11/42*fsup2 + 4/77*fsup4   -  7/462*
       fsup6;
720          eMatrixVal = esub0*E0 + esub1*E1 + esub2*E2 + esub3*E3;
721        )
722      ];
```

```
723       Return[eMatrixVal];
724     )
725   ];
726
727   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.";
728   Options[GenerateElectrostaticTable] = {"Export" -> True, "
        Coefficients" -> "Slater"};
729   GenerateElectrostaticTable[numEmax_Integer:7, OptionsPattern[]]:= (
730     ElectrostaticTable = Table[
731       {numE, SL, SpLp} -> SimplifyFun[Electrostatic[{numE, SL, SpLp},
       "Coefficients" -> OptionValue["Coefficients"]]],
732       {numE, 1, numEmax},
733       {SL, AllowedNKSLTerms[numE]},
734       {SpLp, AllowedNKSLTerms[numE]}
735     ];
736     ElectrostaticTable = Association[Flatten[ElectrostaticTable]];
737     If[OptionValue["Export"],
738       Export[FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"
        }],
739       ElectrostaticTable];
740     ];
741     Return[ElectrostaticTable];
742   );
743
744   (* ###################### Electrostatic ###################### *)
745   (* ########################################################## *)
746
747   (* ########################################################## *)
748   (* ########################## Bases ######################### *)
749
750   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}.";
751   BasisTableGenerator[numE_]:=Module[
752     {energyStatesTable, allowedJ, J, Jp},
753     (
754       energyStatesTable = <||>;
755       allowedJ = AllowedJ[numE];
756       Do[
757       (
758         energyStatesTable[{numE, J}] = EnergyStates[numE, J];
759       ),
760       {Jp, allowedJ},
761       {J,  allowedJ}];
762       Return[energyStatesTable]
763     )
764   ];
765
766   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}.
767 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.
768 ";
769 Options[BasisLSJMJ] = {"AsAssociation" -> False};
770 BasisLSJMJ[numE_, OptionsPattern[]]:=Module[
771   {energyStatesTable, basis, idx1},
772   (
773     energyStatesTable = BasisTableGenerator[numE];
774     basis = Table[
775       energyStatesTable[{numE, AllowedJ[numE][[idx1]]}],
776       {idx1, 1, Length[AllowedJ[numE]]}];
777     basis = Flatten[basis, 1];
778     If[OptionValue["AsAssociation"],
779       (
780         Js    = AllowedJ[numE];
781         basis = Table[(J -> Select[basis, #[[2]] == J &]), {J, Js
    }];
782         basis = Association[basis];
783       )
784     ];
785     Return[basis]
786   )
787 ];
788
789 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}.
790 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.
    ";
792 Options[BasisLSJ]={"AsAssociation"->False};
793 BasisLSJ[numE_,OptionsPattern[]]:=Module[
794   {Js,basis},
795   (
796     Js    = AllowedJ[numE];
797     basis = BasisLSJMJ[numE,"AsAssociation"->False];
798     basis = DeleteDuplicates[{#[[1]],#[[2]]}& /@ basis];
799     If[OptionValue["AsAssociation"],
800       (
801         basis = Association @ Table[(J->Select[basis,#[[2]]==J&]),{
    J,Js}]
802       )
803     ];
804     Return[basis];
805   )
806 ];
807
```

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

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

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

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

NKCFPPhase::usage="Phase between conjugate coefficients of
  fractional parentage according to Nielson and Koster page viii.
  Note that there is a typo on there the expression for zeta should
  be (-1)^((v-1)/2) instead of (-1)^(v - 1/2).";
NKCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
  parentSeniority_, daughterSeniority_]:=Module[
  {spin, orbital, expo, phase},
  (
```

```
849       {spin, orbital} = {1/2, 3};
850       expo = (
851           (parentS + parentL + daughterS + daughterL) -
852           (orbital + spin)
853       );
854       phase = Phaser[-expo];
855       If[parent == 2*orbital,
856           phase = phase * Phaser[(daughterSeniority-1)/2]];
857       Return[phase];
858     )
859   ];
860
861   Options[CFPExpander] = {"Export" -> True, "PhaseFunction" -> "NK"};
862   CFPExpander::usage="Using the coefficients of fractional parentage
        up to f7 this function calculates them up to f14.
863   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.";
864   CFPExpander[OptionsPattern[]]:=Module[
865     {orbital, halfFilled, fullShell, parentMax, PhaseFun,
866     complementaryCFPs, daughter, conjugateDaughter,
867     conjugateParent, parentTerms, daughterTerms,
868     parentCFPs, daughterSeniority, daughterS, daughterL,
869     parentCFP, parentTerm, parentCFPval,
870     parentS, parentL, parentSeniority, phase, prefactor,
871     newCFPval, key, extendedCFPs, exportFname},
872     (
873       orbital    = 3;
874       halfFilled = 2 * orbital + 1;
875       fullShell  = 2 * halfFilled;
876       parentMax  = 2 * orbital;
877
878       PhaseFun  = <|
879           "Judd" -> JuddCFPPhase,
880           "NK" -> NKCFPPhase|>[OptionValue["PhaseFunction"]];
881       PrintTemporary["Calculating CFPs using the phase system from ",
        PhaseFun];
882       (* Initialize everything with lists to be filled in the next Do
        *)
883       complementaryCFPs =
884           Table[
885           ({numE, term} -> {term}),
886           {numE, halfFilled + 1, fullShell - 1, 1},
887           {term, AllowedNKSLTerms[numE]
888           }];
889       complementaryCFPs = Association[Flatten[complementaryCFPs]];
890       Do[(
891           daughter          = parent + 1;
892           conjugateDaughter = fullShell - parent;
893           conjugateParent   = conjugateDaughter - 1;
894           parentTerms       = AllowedNKSLTerms[parent];
895           daughterTerms     = AllowedNKSLTerms[daughter];
```

```
896             Do[
897             (
898                 parentCFPs              = Rest[CFP[{daughter,
    daughterTerm}]];
899                 daughterSeniority       = Seniority[daughterTerm];
900                 {daughterS, daughterL} = FindSL[daughterTerm];
901                 Do[
902                 (
903                     {parentTerm, parentCFPval} = parentCFP;
904                     {parentS, parentL}         = FindSL[parentTerm];
905                     parentSeniority            = Seniority[parentTerm];
906                     phase = PhaseFun[parent, parentS, parentL,
907                                      daughterS, daughterL,
908                                      parentSeniority, daughterSeniority
    ];
909                     prefactor = (daughter * TPO[daughterS, daughterL])
    /
910                                 (conjugateDaughter * TPO[parentS,
    parentL]);
911                     prefactor = Sqrt[prefactor];
912                     newCFPval = phase * prefactor * parentCFPval;
913                     key = {conjugateDaughter, parentTerm};
914                     complementaryCFPs[key] = Append[complementaryCFPs[
    key], {daughterTerm, newCFPval}]
915                 ),
916                 {parentCFP, parentCFPs}
917                 ]
918             ),
919             {daughterTerm, daughterTerms}
920             ]
921             ),
922       {parent, 1, parentMax}
923       ];
924
925       complementaryCFPs[{14, "1S"}] = {"1S", {"2F",1}};
926       extendedCFPs          = Join[CFP, complementaryCFPs];
927       If[OptionValue["Export"];,
928       (
929           exportFname = FileNameJoin[{moduleDir, "data", "
    CFPs_extended.m"}];
930           Print["Exporting to ", exportFname];
931           Export[exportFname, extendedCFPs];
932       )
933       ];
934       Return[extendedCFPs];
935   )
936   ];
937
938   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.
939   The data being parsed here is the file attachment B1F_ALL.TXT which
        comes from Velkov's thesis.";
940   Options[GenerateCFPTable] = {"Export" -> True};
```

```
941  GenerateCFPTable[OptionsPattern[]]:=Module[
942    {rawText, rawLines, leadChar, configIndex, line, daughter,
943    lineParts, numberCode, parsedNumber, toAppend, CFPTablefname},
944    (
945      CleanWhitespace[string_]     := StringReplace[string,
       RegularExpression["\\s+"]->" "];
946      AddSpaceBeforeMinus[string_] := StringReplace[string,
       RegularExpression["(?<!\\s)-"]->" -"];
947      ToIntegerOrString[list_]     := Map[If[StringMatchQ[#,
       NumberString], ToExpression[#], #] &, list];
948      CFPTable      = ConstantArray[{},7];
949      CFPTable[[1]] = {{"2F",{"1S",1}}};
950
951      (* Cleaning before processing is useful *)
952      rawText  = Import[FileNameJoin[{moduleDir, "data", "B1F_ALL.TXT
       "}]];
953      rawLines = StringTrim/@StringSplit[rawText,"\n"];
954      rawLines = Select[rawLines,#!=""&];
955      rawLines = CleanWhitespace/@rawLines;
956      rawLines = AddSpaceBeforeMinus/@rawLines;
957
958      Do[(
959        (* the first character can be used to identify the start of a
        block *)
960        leadChar=StringTake[line,{1}];
961        (* ..FN, N is at position 50 in that line *)
962        If[leadChar=="[",
963        (
964          configIndex=ToExpression[StringTake[line,{50}]];
965          Continue[];
966        )
967        ];
968        (* Identify which daughter term is being listed *)
969        If[StringContainsQ[line,"[DAUGHTER TERM]"],
970          daughter=StringSplit[line,"["][[1]];
971          CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{
       daughter}];
972          Continue[];
973        ];
974        (* Once we get here we are already parsing a row with
       coefficient data *)
975        lineParts    = StringSplit[line," "];
976        parent       = lineParts[[1]];
977        numberCode   = ToIntegerOrString[lineParts[[3;;]]];
978        parsedNumber = SquarePrimeToNormal[numberCode];
979        toAppend     = {parent,parsedNumber};
980        CFPTable[[configIndex]][[-1]] = Append[CFPTable[[configIndex
       ]][[-1]], toAppend]
981      ),
982      {line,rawLines}];
983      If[OptionValue["Export"],
984        (
985        CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"
       }];
986        Export[CFPTablefname, CFPTable];
```

```
        )
      ];
      Return[CFPTable];
    )
  ];

  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.";
  Options[GenerateCFPAssoc] = {"Export" -> True};
  GenerateCFPAssoc[OptionsPattern[]]:= (
    CFPAssoc = Association[];
    Do[
      (daughterTerms = AllowedNKSLTerms[numE];
      parentTerms    = AllowedNKSLTerms[numE - 1];
      Do[
        (
        cfps = CFP[{numE, daughter}];
        cfps = cfps[[2 ;;]];
        parents = First /@ cfps;
        Do[
          (
          key = {numE, daughter, parent};
          cfp = If[
            MemberQ[parents, parent],
            (
              idx = Position[parents, parent][[1, 1]];
              cfps[[idx]][[2]]
            ),
            0
            ];
          CFPAssoc[key] = cfp;
          ),
          {parent, parentTerms}
          ]
        ),
        {daughter, daughterTerms}
        ]
      ),
      {numE, 1, 14}
      ];
    If[OptionValue["Export"],
      (
      CFPAssocfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"
    }];
      Export[CFPAssocfname, CFPAssoc];
      )
    ];
    Return[CFPAssoc];
  );

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

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

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

```

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

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, " "],
```

```
1171            (
1172              multiplierSymbol = StringSplit[header, " "][[1]];
1173              multiplierValue = multiFactorValues[multiplierSymbol];
1174              operatorSymbol = StringSplit[header, " "][[2]];
1175              oppyS = Append[oppyS, operatorSymbol];
1176            ),
1177            (
1178              multiplierValue = 1;
1179              operatorSymbol = header;
1180            )
1181          ];
1182          normalValues = 1/multiplierValue*col[[2 ;;]];
1183          Join[{operatorSymbol}, normalValues], {col, tab1[[3 ;;]]}
1184        ];
1185
1186      (*Create an association for the matrix elements in the f^3 config
         *)
1187      juddOperators = Association[];
1188      Do[(
1189        col      = normalTable[[colIndex]];
1190        opLabel  = col[[1]];
1191        opValues = col[[2 ;;]];
1192        opMatrix = AssociationThread[matrixKeys -> opValues];
1193        Do[(
1194          opMatrix[Reverse[mKey]] = opMatrix[mKey]
1195          ),
1196        {mKey, matrixKeys}
1197        ];
1198        juddOperators[{3, opLabel}] = opMatrix),
1199        {colIndex, 1, Length[normalTable]}
1200      ];
1201
1202      (* special case of t2 in f3 *)
1203      (* this is the same as getting the matrix elements from Judd 1966
          *)
1204      numE = 3;
1205      e3Op      = juddOperators[{3, "e_{3}"}];
1206      t2prime   = juddOperators[{3, "t_{2}^{'}"}];
1207      prefactor = 1/(70 Sqrt[2]);
1208      t2Op = (# -> (t2prime[#] + prefactor*e3Op[#])) & /@ Keys[t2prime
         ];
1209      t2Op = Association[t2Op];
1210      juddOperators[{3, "t_{2}"}] = t2Op;
1211
1212      (*Special case of t11 in f3*)
1213      t11 = juddOperators[{3, "t_{11}"}];
1214      eβprimeOp = juddOperators[{3, "e_{\\beta}^{'}"}];
1215      t11primeOp = (# -> (t11[#] + Sqrt[3/385] eβprimeOp[#])) & /@ Keys
         [t11];
1216      t11primeOp = Association[t11primeOp];
1217      juddOperators[{3, "t_{11}^{'}"}] = t11primeOp;
1218      If[OptionValue["Export"],
1219        (
1220          (*export them*)
1221          PrintTemporary["Exporting ..."];
```

```
1222          exportFname = FileNameJoin[{moduleDir, "data", "juddOperators
      .m"}];
1223          Export[exportFname, juddOperators];
1224        )
1225      ];
1226      Return[juddOperators];
1227    );
1228
1229    GenerateThreeBodyTables::usage="This function generates the matrix
       elements for the three body operators using the coefficients of
       fractional parentage, including those beyond f^7.";
1230    Options[GenerateThreeBodyTables] = {"Export" -> False};
1231    GenerateThreeBodyTables[nmax_Integer : 14, OptionsPattern[]] := (
1232      tiKeys = {"t_{2}", "t_{2}^{'}", "t_{3}", "t_{4}", "t_{6}", "t_{7}
      ",
1233        "t_{8}", "t_{11}", "t_{11}^{'}", "t_{12}", "t_{14}", "t_{15}",
1234        "t_{16}", "t_{17}", "t_{18}", "t_{19}"};
1235      TSymbolsAssoc = AssociationThread[tiKeys -> TSymbols];
1236      juddOperators = ParseJudd1984[];
1237      (* 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. *)
1238      op3MatrixElement[SL_, SpLp_, opSymbol_] := (
1239        jOP = juddOperators[{3, opSymbol}];
1240        key = {SL, SpLp};
1241        val = If[MemberQ[Keys[jOP], key],
1242          jOP[key],
1243          0];
1244        Return[val];
1245      );
1246      (* 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. *)
1247      Options[ti] = {"Fast" -> True};
1248      ti[nE_, SL_, SpLp_, tiKey_, opOrder_ : 3, OptionsPattern[]]:=
      Module[
1249        {
1250          nn, S, L, Sp, Lp,
1251          cfpSL, cfpSpLp,
1252          parentSL, parentSpLp, tnk, tnks
1253        },
1254        (
1255          {S, L}   = FindSL[SL];
1256          {Sp, Lp} = FindSL[SpLp];
1257          fast     = OptionValue["Fast"];
1258          numH = 14 - nE;
1259          If[fast && Not[MemberQ[{"t_{2}","t_{11}"},tiKey]] && nE > 7,
1260            Return[-tktable[{numH, SL, SpLp, tiKey}]]
1261          ];
1262          If[(S == Sp && L == Lp),
1263          (
```

```
1264            cfpSL   = CFP[{nE, SL}];
1265            cfpSpLp = CFP[{nE, SpLp}];
1266            tnks = Table[(
1267                parentSL   = cfpSL[[nn, 1]];
1268                parentSpLp = cfpSpLp[[mm, 1]];
1269                cfpSL[[nn, 2]] * cfpSpLp[[mm, 2]] *
1270                tktable[{nE - 1, parentSL, parentSpLp, tiKey}]
1271                ),
1272                {nn, 2, Length[cfpSL]},
1273                {mm, 2, Length[cfpSpLp]}
1274                ];
1275            tnk = Total[Flatten[tnks]];
1276            ),
1277          tnk = 0;
1278          ];
1279          Return[ nE / (nE - opOrder) * tnk];
1280        )
1281      ];
1282      (*Calculate the matrix elements of t^i for n up to nmax*)
1283      tktable = <||>;
1284      Do[(
1285        Do[(
1286          tkValue = Which[numE <= 2,
1287            (*Initialize n=1,2 with zeros*)
1288            0,
1289            numE == 3,
1290            (*Grab matrix elem in f^3 from Judd 1984*)
1291            SimplifyFun[op3MatrixElement[SL, SpLp, opKey]],
1292            True,
1293            SimplifyFun[ti[numE, SL, SpLp, opKey, If[opKey == "e_{3}",
    2, 3]]]
1294            ];
1295          tktable[{numE, SL, SpLp, opKey}] = tkValue;
1296          ),
1297        {SL, AllowedNKSLTerms[numE]},
1298        {SpLp, AllowedNKSLTerms[numE]},
1299        {opKey, Append[tiKeys, "e_{3}"]}
1300        ];
1301        PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], "
    configuration complete"]];
1302        ),
1303        {numE, 1, nmax}
1304      ];
1305
1306      (* Now use those matrix elements to determine their sum as
    weighted by their corresponding strengths Ti *)
1307      ThreeBodyTable = <||>;
1308      Do[
1309        Do[
1310        (
1311          ThreeBodyTable[{numE, SL, SpLp}] = (
1312            Sum[(
1313              If[tiKey == "t_{2}", t2Switch, 1] *
1314              tktable[{numE, SL, SpLp, tiKey}] *
1315              TSymbolsAssoc[tiKey] +
```

```
1316              If[tiKey == "t_{2}", 1 - t2Switch, 0] *
1317              (-tktable[{14 - numE, SL, SpLp, tiKey}]) *
1318              TSymbolsAssoc[tiKey]
1319              ),
1320            {tiKey, tiKeys}
1321            ]
1322        );
1323      ),
1324      {SL, AllowedNKSLTerms[numE]},
1325      {SpLp, AllowedNKSLTerms[numE]}
1326      ];
1327    PrintTemporary[StringJoin["\[ScriptF]", ToString[numE], " matrix
     complete"]];,
1328    {numE, 1, 7}
1329    ];
1330
1331    ThreeBodyTables = Table[(
1332      terms = AllowedNKSLTerms[numE];
1333      singleThreeBodyTable =
1334        Table[
1335          {SL, SLp} -> ThreeBodyTable[{numE, SL, SLp}],
1336          {SL, terms},
1337          {SLp, terms}
1338        ];
1339      singleThreeBodyTable  = Flatten[singleThreeBodyTable];
1340      singleThreeBodyTables = Table[(
1341          notNullPosition = Position[TSymbols, notNullSymbol][[1,
     1]];
1342          reps = ConstantArray[0, Length[TSymbols]];
1343          reps[[notNullPosition]] = 1;
1344          rep = AssociationThread[TSymbols -> reps];
1345          notNullSymbol -> Association[(singleThreeBodyTable /. rep)]
1346          ),
1347        {notNullSymbol, TSymbols}
1348        ];
1349      singleThreeBodyTables = Association[singleThreeBodyTables];
1350      numE -> singleThreeBodyTables),
1351      {numE, 1, 7}
1352  ];
1353
1354  ThreeBodyTables = Association[ThreeBodyTables];
1355  If[OptionValue["Export"],
1356    (
1357      threeBodyTablefname = FileNameJoin[{moduleDir, "data", "
     ThreeBodyTable.m"}];
1358      Export[threeBodyTablefname, ThreeBodyTable];
1359      threeBodyTablesfname = FileNameJoin[{moduleDir, "data", "
     ThreeBodyTables.m"}];
1360      Export[threeBodyTablesfname, ThreeBodyTables];
1361    )
1362   ];
1363  Return[{ThreeBodyTable, ThreeBodyTables}];
1364  );
1365
1366  ScalarOperatorProduct::usage="ScalarOperatorProduct[op1, op2, numE]
```

```
    calculated the innerproduct between the two scalar operators op1
   and op2.";
1367 ScalarOperatorProduct[op1_, op2_, numE_]:=Module[
1368   {terms, S, L, factor, term1, term2},
1369   (
1370     terms = AllowedNKSLTerms[numE];
1371     Simplify[
1372       Sum[(
1373         {S, L} = FindSL[term1];
1374         factor = TPO[S, L];
1375         factor * op1[{term1, term2}] * op2[{term2, term1}]
1376         ),
1377       {term1, terms},
1378       {term2, terms}
1379       ]
1380     ]
1381   )
1382 ];
1383
1384 (* ################## Three Body Operators ##################### *)
1385 (* ############################################################ *)
1386
1387 (* ############################################################ *)
1388 (* ################### Reduced SOO and ECSO ################### *)
1389
1390 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.
1391 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.
1392 ";
1393 ReducedT11inf2[SL_, SpLp_] := Module[
1394   {T11inf2},
1395   (
1396     T11inf2 = <|
1397       {"1S", "3P"} -> 6 M0 + 2 M2 + 10/11 M4,
1398       {"3P", "3P"} -> -36 M0 - 72 M2 - 900/11 M4,
1399       {"3P", "1D"} -> -Sqrt[(2/15)] (27 M0 + 14 M2 + 115/11 M4),
1400       {"1D", "3F"} -> Sqrt[2/5] (23 M0 + 6 M2 - 195/11 M4),
1401       {"3F", "3F"} -> 2 Sqrt[14] (-15 M0 - M2 + 10/11 M4),
1402       {"3F", "1G"} -> Sqrt[11] (-6 M0 + 64/33 M2 - 1240/363 M4),
1403       {"1G", "3H"} -> Sqrt[2/5] (39 M0 - 728/33 M2 - 3175/363 M4),
1404       {"3H", "3H"} -> 8/Sqrt[55] (-132 M0 + 23 M2 + 130/11 M4),
1405       {"3H", "1I"} -> Sqrt[26] (-5 M0 - 30/11 M2 - 375/1573 M4)
1406       |>;
1407     Which[
1408       MemberQ[Keys[T11inf2],{SL,SpLp}],
1409         Return[T11inf2[{SL,SpLp}]],
1410       MemberQ[Keys[T11inf2],{SpLp,SL}],
1411         Return[T11inf2[{SpLp,SL}]],
1412       True,
1413         Return[0]
1414       ]
```

```
    )
  ];

  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] +
        Reducedt11inf2[SL, SpLp] -
        a13 / 6 * matElement
    );
    redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
    Return[redSOOandECSOinf2];
  )
];

ReducedSOOandECSOinfn::usage="ReducedSOOandECSOinfn[numE, SL, SpLp]
   calculates the reduced matrix elements of the (spin-other-orbit +
   ECSO) operator for the f^numE configuration corresponding to the
  terms SL and SpLp. This is done recursively, starting from
  tabulated values for f^2 from \"Judd, BR, HM Crosswhite, and
  Hannah Crosswhite. \"Intra-Atomic Magnetic Interactions for f
  Electrons.\" Physical Review 169, no. 1 (1968): 130.\", and by
  using equation (4) of that same paper.
";
ReducedSOOandECSOinfn[numE_, SL_, SpLp_]:=Module[
  {spin, orbital, t, S, L, Sp, Lp, idx1, idx2, cfpSL, cfpSpLp,
  parentSL, Sb, Lb, Sbp, Lbp, parentSpLp, funval},
  (
    {spin, orbital} = {1/2, 3};
    {S, L}   = FindSL[SL];
    {Sp, Lp} = FindSL[SpLp];
    t = 1;
    cfpSL    = CFP[{numE, SL}];
    cfpSpLp  = CFP[{numE, SpLp}];
```

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

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

```
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                    {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
    ]], 0.1]|>]},
                    {template3[<|"speed"->Round[QuantityMagnitude[Now
    -startTime, "ms"]/(numiter), 0.01]|>]}, {ProgressIndicator[Dynamic
    [numiter], {1, totalIters}]}
                  },
                  Frame->All
              ],
              Full,
              Alignment->Center
            ]
          ]
        ];
      )
    ];
    SOOandECSOLSTable = <||>;
    numiter   = 1;
    startTime = Now;
    Do[
      (
        numiter+= 1;
        SOOandECSOLSTable[{numE, SL, SpLp}] = Which[
          numE==1,
          0,
          numE==2,
          SimplifyFun[ReducedSOOandECSOinf2[SL, SpLp]],
          True,
          SimplifyFun[ReducedSOOandECSOinfn[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", "
    ReducedSOOandECSOLSTable.m"}];
      Export[fname, SOOandECSOLSTable];
      )
    ];
    Return[SOOandECSOLSTable];
  );

  (* ################## Reduced SOO and ECSO ################## *)
  (* ############################################################ *)

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

  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.
1585  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.
1586  ";
1587  ReducedT22inf2[SL_, SpLp_]:=Module[
1588    {statePosition, PsiPsipStates, m0, m2, m4, Tkk2m},
1589    (
1590      T22inf2 = <|
1591      {"3P", "3P"} -> -12 M0 - 24 M2 - 300/11 M4,
1592      {"3P", "3F"} -> 8/Sqrt[3] (3 M0 + M2 - 100/11 M4),
1593      {"3F", "3F"} -> 4/3 Sqrt[14] (-M0 + 8 M2 - 200/11 M4),
1594      {"3F", "3H"} -> 8/3 Sqrt[11/2] (2 M0 - 23/11 M2 - 325/121 M4),
1595      {"3H", "3H"} -> 4/3 Sqrt[143] (M0 - 34/11 M2 - (1325/1573) M4)
1596      |>;
1597      Which[
1598        MemberQ[Keys[T22inf2],{SL,SpLp}],
1599          Return[T22inf2[{SL,SpLp}]],
1600        MemberQ[Keys[T22inf2],{SpLp,SL}],
1601          Return[T22inf2[{SpLp,SL}]],
1602        True,
1603          Return[0]
1604      ]
1605    )
1606  ];
1607
1608  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.
1609  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.\"
1610  ";
1611  ReducedT22infn[numE_, SL_, SpLp_]:=Module[
1612    {spin, orbital, t, idx1, idx2, S, L, Sp, Lp, cfpSL, cfpSpLp,
    parentSL, parentSpLp, Sb, Lb, Tnkk, phase, Sbp, Lbp},
1613    (
1614      {spin, orbital} = {1/2, 3};
1615      {S, L}   = FindSL[SL];
1616      {Sp, Lp} = FindSL[SpLp];
1617      t = 2;
1618      cfpSL    = CFP[{numE, SL}];
1619      cfpSpLp  = CFP[{numE, SpLp}];
1620      Tnkk = Sum[(
1621        parentSL   = cfpSL[[idx2, 1]];
1622        parentSpLp = cfpSpLp[[idx1, 1]];
1623        {Sb, Lb}   = FindSL[parentSL];
1624        {Sbp, Lbp} = FindSL[parentSpLp];
1625        phase = Phaser[Sb + Lb + spin + orbital + Sp + Lp];
1626        (
1627          phase *
1628          cfpSpLp[[idx1, 2]] * cfpSL[[idx2, 2]] *
1629          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];
    )
  ];

  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];
  );

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

```
              SixJay[{Sp, Lp, J}, {L, S, α}] *
              T22Table[{numE, SL, SpLp}]
            );
      Return[val]
    )
  ];

  GenerateSpinSpinTable::usage="GenerateSpinSpinTable[nmax] generates
     the 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];
    );

  (* ############################################################ *)
  (* ###################### 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.\".
1749 ";
1750 SOOandECSO[numE_, SL_, SpLp_, J_]:=Module[
1751   {S, Sp, L, Lp, α, val},
1752   (
1753     α = 1;
1754     {S, L}   = FindSL[SL];
1755     {Sp, Lp} = FindSL[SpLp];
1756     val = (
1757             Phaser[Sp + L + J] *
1758             SixJay[{Sp, Lp, J}, {L, S, α}] *
1759             SOOandECSOLSTable[{numE, SL, SpLp}]
1760           );
1761     Return[val];
1762   )
1763 ];
1764
1765 Prescaling = {P2 -> P2/225, P4 -> P4/1089, P6 -> 25 * P6 / 184041};
1766
1767 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.";
1768 Options[GenerateSOOandECSOTable] = {"Export"->False}
1769 GenerateSOOandECSOTable[nmax_, OptionsPattern[]]:= (
1770   SOOandECSOTable = <||>;
1771   Do[
1772     SOOandECSOTable[{numE, SL, SpLp, J}] = (SOOandECSO[numE, SL,
    SpLp, J] /. Prescaling);,
1773   {numE, 1, nmax},
1774   {J, MinJ[numE], MaxJ[numE]},
1775   {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
1776   {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
1777   ];
1778   If[OptionValue["Export"],
1779   (
1780     fname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
1781     Export[fname, SOOandECSOTable];
1782   )
1783   ];
1784   Return[SOOandECSOTable];
1785 );
1786
1787 (* ## Spin-Other-Orbit and Electrostatically-Correlated-Spin-Orbit
    ## *)
1788 (* ################################################################ *)
1789
1790 (* ################################################################ *)
1791 (* ################### Magnetic Interactions ################### *)
```

```
1792
1793  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.
1794  The part corresponding to the spin-spin interaction is provided by
      SpinSpin[{numE, SLJ, SLJp, J}].
1795  The part corresponding to SOO and ECSO is provided by the function
      SOOandECSO[{numE, SLJ, SLJp, J}].
1796  The function requires chenDeltas to be loaded into the session.
1797  The option \"ChenDeltas\" can be used to include or exclude the
      Chen deltas from the calculation. The default is to exclude them."
      ;
1798  Options[MagneticInteractions] = {"ChenDeltas" -> False};
1799  MagneticInteractions[{numE_, SLJ_, SLJp_, J_}, OptionsPattern[]] :=
1800    (
1801      key = {numE, SLJ, SLJp, J};
1802      ss = \[Sigma]SS * SpinSpinTable[key];
1803      sooandecso = SOOandECSOTable[key];
1804      total = ss + sooandecso;
1805      total = SimplifyFun[total];
1806      If[
1807        Not[OptionValue["ChenDeltas"]],
1808        Return[total]
1809      ];
1810      (* In the type A errors the wrong values are different *)
1811      If[MemberQ[Keys[chenDeltas["A"]], {numE, SLJ, SLJp}],
1812        (
1813          {S, L} = FindSL[SLJ];
1814          {Sp, Lp} = FindSL[SLJp];
1815          phase   = Phaser[Sp + L + J];
1816          Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
1817          Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
1818          {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SLJ
      , SLJp}]["wrong"];
1819          total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
1820          total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
1821          total  = total /. Prescaling;
1822          total  = wChErrA * total + (1 - wChErrA) * (ss + sooandecso
      )
1823        )
1824      ];
1825      (* In the type B errors the wrong values are zeros all around
      *)
1826      If[MemberQ[chenDeltas["B"], {numE, SLJ, SLJp}],
1827        (
1828          {S, L} = FindSL[SLJ];
1829          {Sp, Lp} = FindSL[SLJp];
1830          phase = Phaser[Sp + L + J];
1831          Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
1832          Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
1833          {M0v, M2v, M4v, P2v, P4v, P6v} = {0, 0, 0, 0, 0, 0};
1834          total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
```

```
1835            total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
1836            total  = total /. Prescaling;
1837            total  = wChErrB * total + (1 - wChErrB) * (ss + sooandecso
    )
1838          )
1839        ];
1840        Return[total];
1841      )
1842
1843    (* ################## Magnetic Interactions ################## *)
1844    (* ########################################################### *)
1845
1846    (* ########################################################### *)
1847    (* ###################### Crystal Field ###################### *)
1848
1849    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.";
1850    Cqk[numE_, q_, k_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_]:=Module[
1851      {S, Sp, L, Lp, orbital, val},
1852      (
1853        orbital = 3;
1854        {S, L}   = FindSL[NKSL];
1855        {Sp, Lp} = FindSL[NKSLp];
1856        f1  = ThreeJay[{J, -M}, {k, q}, {Jp, Mp}];
1857        val =
1858          If[f1==0,
1859            0,
1860            (
1861              f2 = SixJay[{L, J, S}, {Jp, Lp, k}] ;
1862              If[f2==0,
1863                0,
1864                (
1865                  f3 = ReducedUkTable[{numE, orbital, NKSL, NKSLp, k}];
1866                  If[f3==0,
1867                    0,
1868                    (
1869                      (
1870                        Phaser[J - M + S + Lp + J + k] *
1871                        Sqrt[TPO[J, Jp]] *
1872                        f1 *
1873                        f2 *
1874                        f3 *
1875                        Ck[orbital, k]
1876                      )
1877                    )
1878                  ]
1879                )
1880              ]
1881            )
1882          ];
1883        Return[val];
1884      )
1885    ];
1886
```

```
1887  Bqk::usage="Real part of the Bqk coefficients.";
1888  Bqk[q_, 2] := {B02/2, B12, B22}[[q + 1]];
1889  Bqk[q_, 4] := {B04/2, B14, B24, B34, B44}[[q + 1]];
1890  Bqk[q_, 6] := {B06/2, B16, B26, B36, B46, B56, B66}[[q + 1]];
1891
1892  Sqk::usage="Imaginary part of the Bqk coefficients.";
1893  Sqk[q_, 2] := {0, S12, S22}[[q + 1]];
1894  Sqk[q_, 4] := {0, S14, S24, S34, S44}[[q + 1]];
1895  Sqk[q_, 6] := {0, S16, S26, S36, S46, S56, S66}[[q + 1]];
1896
1897  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.
1898  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.";
1899  CrystalField[numE_, NKSL_, J_, M_, NKSLp_, Jp_, Mp_] := (
1900    Sum[
1901      (
1902        cqk  = Cqk[numE,  q, k, NKSL, J, M, NKSLp, Jp, Mp];
1903        cmqk = Cqk[numE, -q, k, NKSL, J, M, NKSLp, Jp, Mp];
1904        Bqk[q, k]   * (cqk + (-1)^q * cmqk) +
1905        I*Sqk[q, k] * (cqk - (-1)^q * cmqk)
1906        ),
1907    {k, {2, 4, 6}},
1908    {q, 0, k}
1909    ]
1910  )
1911
1912  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.";
1913  TotalCFIters[i_, j_] := (
1914    numIters = {196, 8281, 132496, 1002001, 4008004, 9018009,
        11778624};
1915    Return[Total[numIters[[i ;; j]]]];
1916    )
1917
1918  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.";
1919  Options[GenerateCrystalFieldTable] = {"Export" -> False, "Progress"
        -> True, "Compress" -> True}
1920  GenerateCrystalFieldTable[numEs_List:{1,2,3,4,5,6,7},
        OptionsPattern[]]:= (
```

```
    ExportFun =
    If[OptionValue["Compress"],
      ExportMZip,
      Export
    ];
    numiter = 1;
    template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
    template2 = StringTemplate["`remtime` min remaining"];
    template3 = StringTemplate["Iteration speed = `speed` ms/it"];
    template4 = StringTemplate["Time elapsed = `runtime` min"];
    totalIter = Total[TotalCFIters[#, #] & /@ numEs];
    freebies = 0;
    startTime = Now;
    If[And[OptionValue["Progress"], frontEndAvailable],
      progBar = PrintTemporary[
        Dynamic[
          Pane[
            Grid[
              {
                {Superscript["f", numE]},
                {template1[<|"numiter" -> numiter, "totaliter" ->
totalIter|>]},
                {template4[<|"runtime" -> Round[QuantityMagnitude[
UnitConvert[(Now - startTime), "min"]], 0.1]|>]},
                {template2[<|"remtime" -> Round[QuantityMagnitude[
UnitConvert[(Now - startTime)/(numiter - freebies) * (totalIter -
numiter), "min"]], 0.1]|>]},
                {template3[<|"speed" -> Round[QuantityMagnitude[Now -
 startTime, "ms"]/(numiter-freebies), 0.01]|>]},
                {ProgressIndicator[Dynamic[numiter], {1, totalIter}]}
              },
            Frame -> All
            ],
          Full,
          Alignment -> Center
          ]
        ]
      ];
    ];
    Do[
      (
        exportFname = FileNameJoin[{moduleDir, "data", "
CrystalFieldTable_f"<>ToString[numE]<>".m"}];
        If[FileExistsQ[exportFname],
          Print["File exists, skipping ..."];
          numiter+=  TotalCFIters[numE, numE];
          freebies+= TotalCFIters[numE, numE];
          Continue[];
        ];
        CrystalFieldTable = <||>;
        Do[
          (
            numiter+= 1;
            CrystalFieldTable[{numE, NKSL, J, M, NKSLp, Jp, Mp}] =
CrystalField[numE, NKSL, J, M, NKSLp, Jp, Mp];
```

```
          ),
        {J, MinJ[numE], MaxJ[numE]},
        {Jp, MinJ[numE], MaxJ[numE]},
        {M, AllowedMforJ[J]},
        {Mp, AllowedMforJ[Jp]},
        {NKSL , First /@ AllowedNKSLforJTerms[numE, J]},
        {NKSLp, First /@ AllowedNKSLforJTerms[numE, Jp]}
        ];
        If[And[OptionValue["Progress"],frontEndAvailable],
          NotebookDelete[progBar]
        ];
        If[OptionValue["Export"],
          (
            Print["Exporting to file "<>ToString[exportFname]];
            ExportFun[exportFname, CrystalFieldTable];
          )
        ];
      ),
    {numE, numEs}
    ]
  )

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

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

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

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

```
2019
2020  CasimirG2::usage = "CasimirG2[SL, SpLp] returns LS reduced matrix
        element of the configuration interaction term corresponding to the
         Casimir operator of G2.";
2021  CasimirG2[{SL_, SpLp_}] := (
2022    Ulabel = FindNKLSTerm[SL][[1]][[4]];
2023    If[SL==SpLp,
2024      β * GG2U[Ulabel],
2025      0
2026    ]
2027  )
2028
2029  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.
2030  Reference: Wybourne, \"Spectroscopic Properties of Rare Earths\",
        table 2-7.";
2031  GSO7W := Association[
2032    {
2033      "000" -> 0,
2034      "100" -> 3/5,
2035      "110" -> 5/5,
2036      "111" -> 6/5,
2037      "200" -> 7/5,
2038      "210" -> 9/5,
2039      "211" -> 10/5,
2040      "220" -> 12/5,
2041      "221" -> 13/5,
2042      "222" -> 15/5
2043    }
2044  ];
2045
2046  CasimirSO7::usage = "CasimirSO7[SL, SpLp] returns the LS reduced
        matrix element of the configuration interaction term corresponding
         to the Casimir operator of R7.";
2047  CasimirSO7[{SL_, SpLp_}] := (
2048    Wlabel = FindNKLSTerm[SL][[1]][[3]];
2049    If[SL==SpLp,
2050      γ * GSO7W[Wlabel],
2051      0
2052    ]
2053  )
2054
2055  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.";
2056  ElectrostaticConfigInteraction[{SL_, SpLp_}]:=Module[
2057    {S, L, val},
2058    (
2059      {S, L} = FindSL[SL];
2060      val = (
2061        If[SL == SpLp,
2062          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,
    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}]
```

```
2110                )
2111              ];
2112            matValue += CFTable[{numE, SLterm, J, MJ, SpLpterm, Jp, MJp
      }];
2113            matValue,
2114          {NKSLJMp, NKSLJMps},
2115          {NKSLJM , NKSLJMs}
2116          ];
2117     If[OptionValue["Sparse"],
2118       eMatrix = SparseArray[eMatrix]
2119     ];
2120     Return[eMatrix]
2121   )
2122   ];
2123
2124   EnergyStates::usage = "Alias for AllowedNKSLJMforJTerms. At some
        point may be used to redefine states used in basis.";
2125   EnergyStates[numE_, J_]:= AllowedNKSLJMforJTerms[numE, J];
2126
2127   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.";
2128   Options[JJBlockMatrixFileName] = {"FilenameAppendix" -> ""}
2129   JJBlockMatrixFileName[numE_Integer, OptionsPattern[]] := (
2130     fileApp = OptionValue["FilenameAppendix"];
2131     fname = FileNameJoin[{moduleDir,
2132         "hams",
2133         StringJoin[{"f", ToString[numE], "_JJBlockMatrixTable",
        fileApp ,".m"}]}];
2134     Return[fname];
2135   );
2136
2137   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.";
2138   Options[TabulateJJBlockMatrixTable] = {"Sparse"->True, "ChenDeltas"
        ->False};
2139   TabulateJJBlockMatrixTable[numE_, CFTable_, OptionsPattern[]]:= (
2140     JJBlockMatrixTable = <||>;
2141     totalIterations = Length[AllowedJ[numE]]^2;
2142     template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
2143     template2 = StringTemplate["`remtime` min remaining"];
2144     template4 = StringTemplate["Time elapsed = `runtime` min"];
2145     numiter   = 0;
2146     startTime = Now;
2147     If[$FrontEnd =!= Null,
2148       (
```

```
2149        temp = PrintTemporary[
2150          Dynamic[
2151            Grid[
2152              {
2153                {template1[<|"numiter"->numiter, "totaliter"->
    totalIterations|>]},
2154                {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(Max[1,numiter])*(totalIterations-
    numiter), "min"]], 0.1]|>]},
2155                {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
2156                {ProgressIndicator[numiter, {1, totalIterations}]}
2157              }
2158            ]
2159        ]
2160        ];
2161      )
2162    ];
2163    Do[
2164      (
2165        JJBlockMatrixTable[{numE, J, Jp}] = JJBlockMatrix[numE, J, Jp
    , CFTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas" ->
    OptionValue["ChenDeltas"]];
2166        numiter += 1;
2167      ),
2168    {Jp, AllowedJ[numE]},
2169    {J, AllowedJ[numE]}
2170    ];
2171    If[$FrontEnd =!= Null,
2172      NotebookDelete[temp]
2173    ];
2174    Return[JJBlockMatrixTable];
2175  );
2176
2177  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'|>.";
2178  Options[TabulateManyJJBlockMatrixTables] = {"Overwrite"->False, "
      Sparse"->True, "ChenDeltas"->False, "FilenameAppendix"-> "", "
      Compressed" -> False};
2179  TabulateManyJJBlockMatrixTables[ns_, OptionsPattern[]]:= (
2180    overwrite = OptionValue["Overwrite"];
2181    fNames = <||>;
2182    fileApp = OptionValue["FilenameAppendix"];
2183    ExportFun = If[OptionValue["Compressed"], ExportMZip, Export];
2184    Do[
2185      (
2186        CFdataFilename = FileNameJoin[{moduleDir, "data", "
    CrystalFieldTable_f"<>ToString[numE]<>".zip"}];
```

```
2187        PrintTemporary["Importing CrystalFieldTable from ",
      CFdataFilename, " ..."];
2188        CrystalFieldTable = ImportMZip[CFdataFilename];
2189
2190        PrintTemporary["#------- numE = ", numE, " -------#"];
2191        exportFname = JJBlockMatrixFileName[numE, "FilenameAppendix"
      -> fileApp];
2192        fNames[numE] = exportFname;
2193        If[FileExistsQ[exportFname] && Not[overwrite],
2194          Continue[]
2195        ];
2196        JJBlockMatrixTable = TabulateJJBlockMatrixTable[numE,
      CrystalFieldTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas"
      -> OptionValue["ChenDeltas"]];
2197        If[FileExistsQ[exportFname]&&overwrite,
2198          DeleteFile[exportFname]
2199        ];
2200        ExportFun[exportFname, JJBlockMatrixTable];
2201
2202        ClearAll[CrystalFieldTable];
2203      ),
2204    {numE, ns}
2205    ];
2206    Return[fNames];
2207  );
2208
2209  HamMatrixAssembly::usage="HamMatrixAssembly[numE] returns the
      Hamiltonian matrix for the f^n_i configuration. The matrix is
      returned as a SparseArray.
2210  The function admits an optional parameter \"FilenameAppendix\"
      which can be used to modify the filename to which the resulting
      array is exported to.
2211  It also admits an optional parameter \"IncludeZeeman\" which can be
       used to include the Zeeman interaction.
2212  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.
2213  The option \"ReturnInBlocks\" can be use to return the matrix in
      block or flattened form. The default is to return it in flattened
      form.";
2214  Options[HamMatrixAssembly] = {
2215        "FilenameAppendix"->"",
2216        "IncludeZeeman"->False,
2217        "Set t2Switch"->True,
2218        "ReturnInBlocks"->False};
2219  HamMatrixAssembly[nf_, OptionsPattern[]]:=Module[
2220    {numE, ii, jj, howManyJs, Js, blockHam},
2221    (
2222      (*######################################*)
2223      ImportFun = ImportMZip;
2224      (*######################################*)
2225      (*hole-particle equivalence enforcement*)
2226      numE = nf;
2227      allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2,
      T2p,
```

```
2228        T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
2229        α, β, γ, B02, B04, B06, B12, B14, B16,
2230        B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16,
    S22,
2231        S24, S26, S34, S36, S44, S46, S56, S66, T11, T11p, T12, T14,
    T15, T16,
2232        T17, T18, T19, Bx, By, Bz};
2233      params0 = AssociationThread[allVars, allVars];
2234      If[nf > 7,
2235        (
2236          numE = 14 - nf;
2237          params = HoleElectronConjugation[params0];
2238          If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
2239        ),
2240        params = params0;
2241        If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
2242      ];
2243      (* Load symbolic expressions for LS,J,J' energy sub-matrices.
    *)
2244      emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
    OptionValue["FilenameAppendix"]];
2245      JJBlockMatrixTable = ImportFun[emFname];
2246      (*Patch together the entire matrix representation using J,J'
    blocks.*)
2247      PrintTemporary["Patching JJ blocks ..."];
2248      Js         = AllowedJ[numE];
2249      howManyJs = Length[Js];
2250      blockHam   = ConstantArray[0, {howManyJs, howManyJs}];
2251      Do[
2252        blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[
    jj]]}];,
2253      {ii, 1, howManyJs},
2254      {jj, 1, howManyJs}
2255      ];
2256
2257      (* Once the block form is created flatten it *)
2258      If[Not[OptionValue["ReturnInBlocks"]],
2259        (blockHam = ArrayFlatten[blockHam];
2260        blockHam  = ReplaceInSparseArray[blockHam, params];
2261        ),
2262        (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
    ,{2}];)
2263      ];
2264
2265      If[OptionValue["IncludeZeeman"],
2266        (
2267          PrintTemporary["Including Zeeman terms ..."];
2268          {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
    ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
2269          blockHam += - TeslaToKayser * (Bx * magx + By * magy + Bz *
     magz);
2270        )
2271      ];
2272      Return[blockHam];
2273    )
```

```
2274  ];
2275
2276  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.";
2277  Options[SimplerSymbolicHamMatrix]={
2278    "Export"->True,
2279    "PrependToFilename"->"",
2280    "EorF"->"F",
2281    "Overwrite" -> False,
2282    "Return" -> True,
2283    "Set t2Switch" -> False,
2284    "IncludeZeeman" -> False};
2285  SimplerSymbolicHamMatrix[numE_Integer, simplifier_List,
        OptionsPattern[]]:=Module[
2286    {thisHam,fname, fnamemx},
2287    (
2288      If[Not[ValueQ[ElectrostaticTable]],
2289        LoadElectrostatic[]
2290        ];
2291      If[Not[ValueQ[SOOandECSOTable]],
2292        LoadSOOandECSO[]
2293      ];
2294      If[Not[ValueQ[SpinOrbitTable]],
2295        LoadSpinOrbit[]
2296      ];
2297      If[Not[ValueQ[SpinSpinTable]],
2298        LoadSpinSpin[]
2299      ];
2300      If[Not[ValueQ[ThreeBodyTable]],
2301        LoadThreeBody[]
2302      ];
2303
2304      fname = FileNameJoin[{moduleDir,"hams",OptionValue["
        PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".m"}];
2305      fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
        PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".mx"}];
2306      If[Or[FileExistsQ[fname], FileExistsQ[fnamemx]] && Not[
        OptionValue["Overwrite"]],
2307        (
2308          If[OptionValue["Return"],
2309            (
2310              Which[
2311                FileExistsQ[fnamemx],
2312                (
2313                  Print["File ",fnamemx," already exists, and option
```

```
    \"Overwrite\" is set to False, loading file ..."];
2314                thisHam = Import[fnamemx];
2315                Return[thisHam];
2316              ),
2317              FileExistsQ[fname],
2318              (
2319                Print["File ",fname," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
2320                thisHam = Import[fname];
2321                Print["Exporting to file ",fnamemx, " for quicker
    loading."];
2322                Export[fnamemx,thisHam];
2323                Return[thisHam];
2324              )
2325            ]
2326          ),
2327          (
2328            Print["File ",fname," already exists, skipping ..."];
2329            Return[Null];
2330          )
2331        ]
2332      )
2333    ];
2334
2335    thisHam = HamMatrixAssembly[numE, "Set t2Switch" -> OptionValue
    ["Set t2Switch"], "IncludeZeeman"->OptionValue["IncludeZeeman"]];
2336    thisHam = ReplaceInSparseArray[thisHam, simplifier];
2337    (* This removes zero entries from being included in the sparse
    array *)
2338    thisHam = SparseArray[thisHam];
2339    If[OptionValue["Export"],
2340    (
2341      Print["Exporting to file ",fname, " and to ", fnamemx];
2342      Export[fname,thisHam];
2343      Export[fnamemx,thisHam];
2344    )
2345    ];
2346    If[OptionValue["Return"],
2347      Return[thisHam],
2348      Return[Null]
2349    ];
2350  )
2351 ];
2352
2353
2354 (* ############################################################ *)
2355 (* ################ To Intermediate Coupling ################## *)
2356
2357 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.";
2358 FreeHam[JJBlocks_List, numE_Integer]:=Module[
2359   {Js, basisJ, pivot, freeHam, idx, J, thisJbasis,
    shrunkBasisPositions, theBlock},
2360   (
```

```
      Js      = AllowedJ[numE];
      basisJ  = BasisLSJMJ[numE, "AsAssociation" -> True];
      pivot   = If[OddQ[numE], 1/2, 0];
      freeHam = Table[(
        J          = Js[[idx]];
        theBlock   = JJBlocks[[idx]];
        thisJbasis = basisJ[J];
        (* find the basis vectors that end with pivot *)
        shrunkBasisPositions = Flatten[Position[thisJbasis, {_ ..., pivot}]];
        (* take only those rows and columns *)
        theBlock[[shrunkBasisPositions, shrunkBasisPositions]]
      ),
      {idx, 1, Length[Js]}
      ];
      Return[freeHam];
    )
  ];

  ListRepeater::usage="ListRepeater[list, reps] repeats each element of list reps times.";
  ListRepeater[list_List, repeats_Integer] := (
    Flatten[ConstantArray[#, repeats] & /@ list]
  );

  ListLever::usage="ListLever[vecs, multiplicity] takes a list of vectors and returns all interleaved shifted versions of them.";
  ListLever[vecs_, multiplicity_]:=Module[
  {uppityVecs, uppityVec},
  (
    uppityVecs = Table[(
      uppityVec = PadRight[{#}, multiplicity] & /@ vec;
      uppityVec = Permutations /@ uppityVec;
      uppityVec = Transpose[uppityVec];
      uppityVec = Flatten /@ uppityVec
      ),
    {vec, vecs}
    ];
    Return[Flatten[uppityVecs, 1]];
  )
  ];

  EigenLever::usage="EigenLever[eigenSys, multiplicity] takes a list eigenSys of the form {eigenvalues, eigenvectors} and returns the eigenvalues repeated multiplicity times and the eigenvectors interleaved and shifted accordingly.";
  EigenLever[eigenSys_, multiplicity_]:=Module[
    {eigenVals, eigenVecs, leveledEigenVecs, leveledEigenVals},
    (
      {eigenVals, eigenVecs} = eigenSys;
      leveledEigenVals       = ListRepeater[eigenVals, multiplicity];
      leveledEigenVecs       = ListLever[eigenVecs, multiplicity];
      Return[{Flatten[leveledEigenVals], leveledEigenVecs}]
    )
  ];
```

```
2410
2411
2412  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.
2413  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.
2414  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.
2415  The option \"PrependToFilename\" can be used to append a string to
       the filename to which the function may export to.
2416  The option \"Return\" can be used to choose whether the function
       returns the matrix or not.
2417  The option \"Overwrite\" can be used to overwrite the file if it
       already exists.";
2418  Options[SimplerSymbolicIntermediateHamMatrix] = {
2419    "Export" -> True,
2420    "PrependToFilename" -> "",
2421    "Overwrite" -> False,
2422    "Return" -> True,
2423    "Simplifier" -> Join[
2424      {F0,\[Sigma]SS},
2425      cfSymbols,
2426      TSymbols,
2427      casimirSymbols,
2428      pseudoMagneticSymbols,
2429      marvinSymbols,
2430      DeleteCases[magneticSymbols,ζ]
2431    ]
2432  };
2433  SimplerSymbolicIntermediateHamMatrix[numE_Integer, OptionsPattern
       []] := Module[
2434    {thisHamAssoc, Js, fname, fnamemx, hash, simplifier},
2435    (
2436      simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
2437      hash       = Hash[simplifier];
2438      If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
2439      If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
2440      If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
2441      If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
2442      If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
2443      fname    = FileNameJoin[{moduleDir,"hams",OptionValue["
       PrependToFilename"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
```

```
    numE]<>"-"<>ToString[hash]<>".m"}];
2444    fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Intermediate-SymbolicMatrix-f"<>ToString[
    numE]<>"-"<>ToString[hash]<>".mx"}];
2445    If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue
    ["Overwrite"]],
2446    (
2447      If[OptionValue["Return"],
2448      (
2449        Which[FileExistsQ[fnamemx],
2450        (
2451          Print["File ",fnamemx," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
2452          thisHamAssoc=Import[fnamemx];
2453          Return[thisHamAssoc];
2454        ),
2455        FileExistsQ[fname],
2456        (
2457          Print["File ",fname," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
2458          thisHamAssoc=Import[fname];
2459          Print["Exporting to file ",fnamemx," for quicker loading.
    "];
2460          Export[fnamemx,thisHamAssoc];
2461          Return[thisHamAssoc];
2462        )
2463        ]
2464      ),
2465      (
2466        Print["File ",fname," already exists, skipping ..."];
2467        Return[Null];
2468      )
2469      ]
2470    )
2471    ];
2472    Js            = AllowedJ[numE];
2473    thisHamAssoc = HamMatrixAssembly[numE,
2474      "Set t2Switch"->True,
2475      "IncludeZeeman"->False,
2476      "ReturnInBlocks"->True
2477    ];
2478    thisHamAssoc = Diagonal[thisHamAssoc];
2479    thisHamAssoc = Map[SparseArray[ReplaceInSparseArray[#,
    simplifier]]&,thisHamAssoc,{1}];
2480    thisHamAssoc = FreeHam[thisHamAssoc,numE];
2481    thisHamAssoc = AssociationThread[Js->thisHamAssoc];
2482    If[OptionValue["Export"],
2483      (
2484        Print["Exporting to file ",fname," and to ",fnamemx];
2485        Export[fname,thisHamAssoc];
2486        Export[fnamemx,thisHamAssoc];
2487      )
2488    ];
2489    If[OptionValue["Return"],
2490      Return[thisHamAssoc],
```

```
2491          Return[Null]
2492        ];
2493      )
2494    ];
2495
2496    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.
2497    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.
2498    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.
2499    The following options are admitted:
2500    - \"Overwrite Hamiltonian\", if set to True the function will
        overwrite the symbolic Hamiltonian. Default is False.
2501    - \"Return as states\", see description above. Default is True.
2502    - \"Simplifier\", this is a list with symbols that are set to zero
        for defining the parameters kept in the intermediate coupling
        description.
2503    ";
2504    Options[IntermediateSolver] = {
2505      "Overwrite Hamiltonian" -> False,
2506      "Return as states" -> True,
2507      "Simplifier" -> Join[
2508        cfSymbols,
2509        TSymbols,
2510        casimirSymbols,
2511        pseudoMagneticSymbols,
2512        marvinSymbols,
2513        DeleteCases[magneticSymbols,ζ]
2514      ],
2515      "PrintFun" -> PrintTemporary
2516    };
2517    IntermediateSolver[numE_Integer, params0_Association,
        OptionsPattern[]] := Module[
2518      {ln, simplifier, simpleHam, basis, numHam, eigensys, startTime,
      endTime, diagonalTime, params=params0, globalBasis, eigenVectors,
      eigenEnergies, eigenJs, states, groundEnergy, allEnergies,
      PrintFun},
```

```
2519      (
2520        ln    = theLanthanides[[numE]];
2521        basis = BasisLSJ[numE, "AsAssociation"->True];
2522        simplifier = OptionValue["Simplifier"];
2523        PrintFun   = OptionValue["PrintFun"];
2524        PrintFun["> IntermediateSolver for ",ln," with ",numE," f-
          electrons."];
2525        PrintFun["> Loading the symbolic intermediate coupling
          Hamiltonian ..."];
2526        simpleHam  = SimplerSymbolicIntermediateHamMatrix[numE,
2527          "Simplifier" -> simplifier,
2528          "Overwrite"  -> OptionValue["Overwrite Hamiltonian"]
2529        ];
2530        (* Everything that is not given is set to zero *)
2531        PrintFun["> Setting to zero every parameter not given ..."];
2532        params    = ParamPad[params,"Print"->True];
2533        PrintFun[params];
2534        (* Create the numeric hamiltonian *)
2535        PrintFun["> Replacing parameters in the J-blocks of the
          intermediate coupling Hamiltonian to produce numeric arrays ..."];
2536        numHam    = N /@ Map[ReplaceInSparseArray[#, params]&,
          simpleHam];
2537        Clear[simpleHam];
2538        (* Eigensolver *)
2539        PrintFun["> Diagonalizing the numerical Hamiltonian within each
           separat J-subspace ..."];
2540        startTime    = Now;
2541        eigensys     = Eigensystem /@ numHam;
2542        endTime      = Now;
2543        diagonalTime = QuantityMagnitude[endTime-startTime,"Seconds"];
2544        allEnergies  = Flatten[First/@Values[eigensys]];
2545        groundEnergy = Min[allEnergies];
2546        eigensys     = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys
          ];
2547        eigensys     = Association@KeyValueMap[#1->{basis[#1],#2} &,
          eigensys];
2548        PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
2549        If[OptionValue["Return as states"],
2550          (
2551            PrintFun["> Padding the eigenvectors to correspond to the
              global intermediate basis ..."];
2552            eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values
              [#[[2, 2]] & /@ eigensys]];
2553            globalBasis   = Flatten[Values[basis], 1];
2554            eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
2555            eigenJs       = Flatten[KeyValueMap[ConstantArray[#1,
              Length[#2[[2, 2]]]] &, eigensys]];
2556            states        = Transpose[{eigenEnergies, eigenJs,
              eigenVectors}];
2557            states        = SortBy[states, First];
2558            Return[{globalBasis, states}];
2559          ),
2560          Return[{basis, eigensys}]
2561        ];
2562      )
```

```
];


(* ################ To Intermediate Coupling ################ *)
(* ########################################################## *)

(* ##################### Block assembly ##################### *)
(* ########################################################## *)

(* ########################################################## *)
(* ##################### Optical Operators ################## *)

magOp = <||>;

JJBlockMagDip::usage="JJBlockMagDip[numE, J, Jp] returns the LSJ-
  reduced matrix element of the magnetic dipole operator between the
   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,
  braMJ,    ketMJ,
  matValue, magMatrix},
  (
    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' \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");
```

```
2655        exportFname  = JJBlockMatrixFileName[numE,"FilenameAppendix"->
    appendTo];
2656        fnames[numE] = exportFname;
2657        If[FileExistsQ[exportFname]&&Not[OptionValue["Overwrite"]],
2658          Continue[]
2659        ];
2660        JJBlockMatrixTable = TabulateJJBlockMagDipTable[numE];
2661        If[FileExistsQ[exportFname]&&OptionValue["Overwrite"],
2662          DeleteFile[exportFname]
2663        ];
2664        ExportFun[exportFname,  JJBlockMatrixTable];
2665      ),
2666      {numE,ns}
2667      ];
2668      Return[fnames];
2669    );
2670
2671    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 use to return the matrices in
      blocks. The default is to return the matrices in flattened form.";
2672    Options[MagDipoleMatrixAssembly]={
2673      "FilenameAppendix"->"",
2674      "ReturnInBlocks"->False};
2675    MagDipoleMatrixAssembly[nf_Integer, OptionsPattern[]]:=Module[
2676      {ImportFun, numE, appendTo, emFname, JJBlockMagDipTable,
2677      Js, howManyJs, blockOp, rowIdx, colIdx},
2678      (
2679        ImportFun = ImportMZip;
2680        numE      = nf;
2681        numH      = 14 - numE;
2682        numE      = Min[numE,numH];
2683
2684        appendTo  = (OptionValue["FilenameAppendix"]<>"-magDip");
2685        emFname   = JJBlockMatrixFileName[numE,"FilenameAppendix"->
    appendTo];
2686        JJBlockMagDipTable = ImportFun[emFname];
2687
2688        Js        = AllowedJ[numE];
2689        howManyJs = Length[Js];
2690        blockOp   = ConstantArray[0,{howManyJs,howManyJs}];
2691        Do[
2692          blockOp[[rowIdx,colIdx]] = JJBlockMagDipTable[{numE,Js[[
    rowIdx]],Js[[colIdx]]}],
2693          {rowIdx,1,howManyJs},
2694          {colIdx,1,howManyJs}
2695        ];
2696        If[OptionValue["ReturnInBlocks"],
2697          (
```

```
2698            opMinus = Map[#[[1]]&, blockOp, {4}];
2699            opZero  = Map[#[[2]]&, blockOp, {4}];
2700            opPlus  = Map[#[[3]]&, blockOp, {4}];
2701            opX =   (opMinus - opPlus)/Sqrt[2];
2702            opY = I (opPlus + opMinus)/Sqrt[2];
2703            opZ = opZero;
2704          ),
2705            blockOp = ArrayFlatten[blockOp];
2706            opMinus = blockOp[[;; , ;; , 1]];
2707            opZero  = blockOp[[;; , ;; , 2]];
2708            opPlus  = blockOp[[;; , ;; , 3]];
2709            opX =   (opMinus - opPlus)/Sqrt[2];
2710            opY = I (opPlus + opMinus)/Sqrt[2];
2711            opZ = opZero;
2712        ];
2713        Return[{opX, opY, opZ}];
2714      )
2715    ];
2716
2717    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.
2718    The option \"Units\" can be set to either \"SI\" (so that the units
        of the returned array are A/m^2) or to \"Hartree\".
2719    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.
2720    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 |i> and |j>.";
2721    Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI"
       , "States" -> All};
2722    MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern
       []]:=Module[
2723      {allEigenvecs, Sx, Sy, Sz, Stot ,factor},
2724      (
2725        numE = Min[14-numE0, numE0];
2726        (*If not loaded then load it, *)
2727        If[Or[
2728            Not[MemberQ[Keys[magOp], numE]],
2729            OptionValue["Reload MagOp"]],
2730        (
2731          magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
       MagDipoleMatrixAssembly[numE];
2732        )
2733        ];
2734        allEigenvecs = Transpose[Last /@ theEigensys];
2735        Which[OptionValue["States"] === All,
2736          (
2737            {Sx,Sy,Sz}    = (ConjugateTranspose[allEigenvecs].#.
       allEigenvecs) & /@ magOp[numE];
```

```
2738            Stot          = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2739          ),
2740          IntegerQ[OptionValue["States"]],
2741          (
2742            singleState   = theEigensys[[OptionValue["States"],2]];
2743            {Sx,Sy,Sz}    = (ConjugateTranspose[allEigenvecs].#.
    singleState) & /@ magOp[numE];
2744            Stot          = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2745          )
2746        ];
2747        Which[
2748          OptionValue["Units"] == "SI",
2749            Return[4 \[Mu]B^2 * Stot],
2750          OptionValue["Units"] == "Hartree",
2751            Return[Stot],
2752          True,
2753          (
2754            Print["Invalid option for \"Units\". Options are \"SI\" and
     \"Hartree\"."];
2755            Abort[];
2756          )
2757        ];
2758      )
2759    ];
2760
2761    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.
2762  Based on table 7.3 of Thorne 1999, using g2=1.
2763  The energy unit assumed in eigenSys is kayser.
2764  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>.
2765  By default this assumes that the refractive index is unity, this
      may be changed by setting the option \"RefractiveIndex\" to the
      desired value.
2766  The option \"Lifetime\" can be used to return the reciprocal of the
       transition rates. The default is to return the transition rates."
      ;
2767  Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False, "
      RefractiveIndex"->1};
2768  MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]]:=
      Module[
2769    {AMD,Stot,eigenEnergies,transitionWaveLengthsInMeters,nRefractive
      },
2770    (
2771      nRefractive   = OptionValue["RefractiveIndex"];
2772      numE          = Min[14-numE0, numE0];
2773      Stot          = MagDipLineStrength[eigenSys, numE, "Units"->
```

```
    OptionValue["Units"]];
2774      eigenEnergies = Chop[First/@eigenSys];
2775      energyDiffs   = Outer[Subtract,eigenEnergies,eigenEnergies];
2776      energyDiffs   = ReplaceDiagonal[energyDiffs, Indeterminate];
2777      (* Energies assumed in pseudo-energy unit kayser.*)
2778      transitionWaveLengthsInMeters = 0.01/energyDiffs;
2779
2780      unitFactor    = Which[
2781      OptionValue["Units"]=="Hartree",
2782      (
2783        (* The bohrRadius factor in SI neede to convert the
    wavelengths which are assumed in m*)
2784        16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckFine)) * bohrRadius^3
2785      ),
2786      OptionValue["Units"]=="SI",
2787      (
2788        16 \[Pi]^3 \[Mu]0/(3 hPlanck)
2789      ),
2790      True,
2791      (
2792        Print["Invalid option for \"Units\". Options are \"SI\" and \
    "Hartree\"."];
2793        Abort[];
2794      )
2795      ];
2796      AMD = unitFactor / transitionWaveLengthsInMeters^3 * Stot *
    nRefractive^3;
2797      Which[OptionValue["Lifetime"],
2798        Return[1/AMD],
2799        True,
2800        Return[AMD]
2801      ]
2802    )
2803  ];
2804
2805  GroundStateOscillatorStrength::usage="GroundStateOscillatorStrength
      [eigenSys, numE] calculates the oscillator strengths between the
      ground state and the excited states as given by eigenSys.
2806  Based on equation 8 of Carnall 1965, removing the 2J+1 factor since
       this degeneracy has been removed by the crystal field.
2807  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.
2808  The energy unit assumed in eigenSys is Kayser.
2809  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>.
2810  By default this assumes that the refractive index is unity, this
      may be changed by setting the option \"RefractiveIndex\" to the
      desired value.";
2811  Options[GroundStateOscillatorStrength]={"RefractiveIndex"->1};
2812  GroundStateOscillatorStrength[eigenSys_List, numE_Integer,
      OptionsPattern[]]:=Module[
2813    {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];
    )
  ];

  (* ##################### Optical Operators ##################### *)
  (* ############################################################## *)

  (* ############################################################## *)
  (* ##################### Printers and Labels ##################### *)

  PrintL::usage = "PrintL[L] give the string representation of a given angular momentum.";
  PrintL[L_] := If[StringQ[L], L, StringTake[specAlphabet, {L + 1}]]

  FindSL::usage = "FindSL[LS] gives the spin and orbital angular momentum that corresponds to the provided string LS.";
  FindSL[SL_]:= (
    FindSL[SL] =
    If[StringQ[SL],
      {
        (ToExpression[StringTake[SL, 1]]-1)/2,
        StringPosition[specAlphabet, StringTake[SL, {2}]][[1, 1]]-1
      },
      SL
    ]
  );

  PrintSLJ::usage = "Given a list with three elements {S, L, J} this function returns a symbol where the spin multiplicity is presented as a superscript, the orbital angular momentum as its corresponding spectroscopic letter, and J as a subscript. Function does not check to see if the given J is compatible with the given S and L.";
  PrintSLJ[SLJ_] :=
    RowBox[{SuperscriptBox[" ", 2 SLJ[[1]] + 1],
      SubscriptBox[PrintL[SLJ[[2]]], SLJ[[3]]]}] // DisplayForm;

  PrintSLJM::usage = "Given a list with four elements {S, L, J, MJ} this function returns a symbol where the spin multiplicity is presented as a superscript, the orbital angular momentum as its corresponding spectroscopic letter, and {J, MJ} as a subscript. No attempt is made to guarantee that the given input is consistent."
  ;
```

```
2855  PrintSLJM[SLJM_] :=
2856    RowBox[{SuperscriptBox[" ", 2 SLJM[[1]] + 1],
2857      SubscriptBox[PrintL[SLJM[[2]]], {SLJM[[3]], SLJM[[4]]}]}] //
2858    DisplayForm;
2859
2860  (* ################### Printers and Labels ################### *)
2861  (* ############################################################ *)
2862
2863  (* ############################################################ *)
2864  (* ###################### Term management ###################### *)
2865
2866  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.";
2867  AllowedSLTerms[numE_] := Map[FindSL[First[#]] &, CFPTerms[Min[numE,
        14-numE]]];
2868
2869  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.";
2870  AllowedNKSLTerms[numE_] := Map[First, CFPTerms[Min[numE, 14-numE
       ]]];
2871  AllowedNKSLTerms[0]  = {"1S"};
2872  AllowedNKSLTerms[14] = {"1S"};
2873
2874  MaxJ::usage = "MaxJ[numE] gives the maximum J = S+L that
       corresponds to the configuration f^numE.";
2875  MaxJ[numE_] := Max[Map[Total, AllowedSLTerms[Min[numE, 14-numE]]]];
2876
2877  MinJ::usage = "MinJ[numE] gives the minimum J = S+L that
       corresponds to the configuration f^numE.";
2878  MinJ[numE_] := Min[Map[Abs[Part[#, 1] - Part[#, 2]] &,
       AllowedSLTerms[Min[numE, 14-numE]]]]
2879
2880  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.";
2881  AllowedSLJTerms[numE_] := Module[
2882    {idx1, allowedSL, allowedSLJ},
2883    (
2884      allowedSL = AllowedSLTerms[numE];
2885      allowedSLJ = {};
2886      For[
2887        idx1 = 1,
2888        idx1 <= Length[allowedSL],
2889        termSL = allowedSL[[idx1]];
2890        termsSLJ =
2891          Table[
2892            {termSL[[1]], termSL[[2]], J},
2893          {J,Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
```

```
2894            ];
2895          allowedSLJ = Join[allowedSLJ, termsSLJ];
2896          idx1++
2897        ];
2898        SortBy[allowedSLJ, Last]
2899      )
2900    ];
2901
2902    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.";
2903    AllowedNKSLJTerms[numE_] := Module[
2904      {allowedSL, allowedNKSL, allowedSLJ, nn},
2905      (
2906        allowedNKSL = AllowedNKSLTerms[numE];
2907        allowedSL = AllowedSLTerms[numE];
2908        allowedSLJ = {};
2909        For[
2910          nn = 1,
2911          nn <= Length[allowedSL],
2912          (
2913            termSL = allowedSL[[nn]];
2914            termNKSL = allowedNKSL[[nn]];
2915            termsSLJ =
2916              Table[{termNKSL, J},
2917              {J, Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
2918              ];
2919            allowedSLJ = Join[allowedSLJ, termsSLJ];
2920            nn++
2921          )
2922        ];
2923        SortBy[allowedSLJ, Last]
2924      )
2925    ];
2926
2927    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.";
2928    AllowedNKSLforJTerms[numE_, J_]:=Module[
2929      {allowedSL, allowedNKSL, allowedSLJ, nn, termSL, termNKSL,
          termsSLJ},
2930      (
2931        allowedNKSL = AllowedNKSLTerms[numE];
2932        allowedSL = AllowedSLTerms[numE];
2933        allowedSLJ = {};
2934        For[
2935          nn = 1,
2936          nn <= Length[allowedSL],
2937          (
2938            termSL = allowedSL[[nn]];
2939            termNKSL = allowedNKSL[[nn]];
2940            termsSLJ = If[Abs[termSL[[1]] - termSL[[2]]] <= J <= Total[
```

```
    termSL],
2941            {{termNKSL, J}},
2942            {}
2943            ];
2944          allowedSLJ = Join[allowedSLJ, termsSLJ];
2945          nn++
2946        )
2947      ];
2948      Return[allowedSLJ]
2949    )
2950  ];
2951
2952  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
       }.";
2953  AllowedSLJMTerms[numE_]:=Module[
2954    {allowedSLJ, allowedSLJM, termSLJ, termsSLJM, nn},
2955    (
2956    allowedSLJ = AllowedSLJTerms[numE];
2957    allowedSLJM = {};
2958    For[
2959      nn = 1,
2960      nn <= Length[allowedSLJ],
2961      nn++,
2962      (
2963        termSLJ = allowedSLJ[[nn]];
2964        termsSLJM =
2965          Table[{termSLJ[[1]], termSLJ[[2]], termSLJ[[3]], M},
2966          {M, - termSLJ[[3]], termSLJ[[3]]}
2967          ];
2968        allowedSLJM = Join[allowedSLJM, termsSLJM];
2969      )
2970    ];
2971    Return[SortBy[allowedSLJM, Last]];
2972    )
2973  ];
2974
2975  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}.";
2976  AllowedNKSLJMforJMTerms[numE_, J_, MJ_] := Module[
2977    {allowedSL, allowedNKSL, allowedSLJM, nn},
2978    (
2979      allowedNKSL = AllowedNKSLTerms[numE];
2980      allowedSL   = AllowedSLTerms[numE];
2981      allowedSLJM = {};
2982      For[
2983        nn = 1,
2984        nn <= Length[allowedSL],
2985        termSL = allowedSL[[nn]];
2986        termNKSL = allowedNKSL[[nn]];
2987        termsSLJ = If[(Abs[termSL[[1]] - termSL[[2]]]
```

```
                        <= J
                        <= Total[termSL]
                        && (Abs[MJ] <= J)
                        ),
                        {{termNKSL, J, MJ}},
                        {}];
        allowedSLJM = Join[allowedSLJM, termsSLJ];
        nn++
      ];
      Return[allowedSLJM];
    )
  ];

  AllowedNKSLJMforJTerms::usage = "AllowedNKSLJMforJTerms[numE, J]
    returns a list with all the states that have a total angular
    momentum J. The returned list has elements of the form {{SL (
    string in spectroscopic notation), J}, MJ}, 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}.";
  AllowedNKSLJMforJTerms[numE_, J_] := Module[
    {MJs, labelsAndMomenta, termsWithJ},
    (
      MJs = AllowedMforJ[J];
      (* Pair LS labels and their {S,L} momenta *)
      labelsAndMomenta = ({#, FindSL[#]}) & /@ AllowedNKSLTerms[numE
    ];
      (* A given term will contain J if |L-S|<=J<=L+S *)
      ContainsJ[{SL_String, {S_, L_}}] := (Abs[S - L] <= J <= (S + L)
    );
      (* Keep just the terms that satisfy this condition *)
      termsWithJ = Select[labelsAndMomenta, ContainsJ];
      (* We don't want to keep the {S,L} *)
      termsWithJ = {#[[1]], J} & /@ termsWithJ;
      (* This is just a quick way of including up all the MJ values
    *)
      Return[Flatten /@ Tuples[{termsWithJ, MJs}]]
    )
  ];

  AllowedMforJ::usage = "AllowedMforJ[J] is shorthand for Range[-J, J
    , 1].";
  AllowedMforJ[J_] := Range[-J, J, 1];

  AllowedJ::usage = "AllowedJ[numE] returns the total angular momenta
     J that appear in the f^numE configuration.";
  AllowedJ[numE_] := Table[J, {J, MinJ[numE], MaxJ[numE]}];

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

  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}.";
3029  FindNKLSTerm[SL_]:=Module[
3030    {NKterms, n},
3031    (
3032      n = 7;
3033      NKterms = {{}};
3034      Map[
3035        If[! StringFreeQ[First[#], SL],
3036          If[ToExpression[Part[#, 2]] <= n,
3037            NKterms = Join[NKterms, {#}, 1]
3038          ]
3039        ] &,
3040      fnTermLabels
3041      ];
3042      NKterms = DeleteCases[NKterms, {}];
3043      NKterms
3044    )
3045  ];
3046
3047  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}.";
3048  Options[ParseTermLabels] = {"Export" -> True};
3049  ParseTermLabels[OptionsPattern[]]:=Module[
3050    {labelsTextData, fNtextLabels, nielsonKosterLabels, seniorities,
      RacahW, RacahU},
3051    (
3052      labelsTextData = FileNameJoin[{moduleDir, "data", "
      NielsonKosterLabels_f6_f7.txt"}];
3053      fNtextLabels   = Import[labelsTextData];
3054      nielsonKosterLabels = Partition[StringSplit[fNtextLabels], 3];
3055      termLabels = Map[Part[#, {1}] &, nielsonKosterLabels];
3056      seniorities = Map[ToExpression[Part[# , {2}]] &,
      nielsonKosterLabels];
3057      racahW =
3058        Map[
3059          StringTake[
3060            Flatten[StringCases[Part[# , {3}],
3061              "(" ~~ DigitCharacter ~~ DigitCharacter ~~
      DigitCharacter ~~ ")"]],
3062            {2, 4}
3063          ] &,
3064        nielsonKosterLabels];
3065      racahU =
3066        Map[
3067          StringTake[
3068            Flatten[StringCases[Part[# , {3}],
3069              "(" ~~ DigitCharacter ~~ DigitCharacter ~~ ")"]],
3070            {2, 3}
3071          ] &,
3072        nielsonKosterLabels];
3073      fnTermLabels = Join[termLabels, seniorities, racahW, racahU,
      2];
3074      fnTermLabels = Sort[fnTermLabels];
```

```
3075        If[OptionValue["Export"],
3076          (
3077            broadFname = FileNameJoin[{moduleDir,"data","fnTerms.m"}];
3078            Export[broadFname, fnTermLabels];
3079          )
3080        ];
3081        Return[fnTermLabels];
3082      )
3083    ];
3084
3085    (* ###################### Term management ###################### *)
3086    (* ############################################################ *)
3087
3088    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.";
3089    Options[LoadParameters] = {
3090        "Source"->"Carnall",
3091        "Free Ion"->False,
3092        "gs"->2.002319304386,
3093        "With Uncertainties"->False
3094        };
3095    LoadParameters[Ln_String, OptionsPattern[]]:= Module[
3096      {source, params, uncertain, uncertainKeys, uncertainRules},
3097      (
3098        source = OptionValue["Source"];
3099        params = Which[source=="Carnall",
3100                 (Association[Carnall["data"][Ln]])
3101                 ];
3102        (*If a free ion then all the parameters from the crystal field
        are set to zero*)
3103        If[OptionValue["Free Ion"],
3104          Do[params[cfSymbol] = 0, {cfSymbol, cfSymbols}]
3105        ];
3106        params[F0] = 0;
3107        params[M2] = 0.56*params[M0]; (*See Carnall 1989,Table I,
        caption,probably fixed based on HF values*)
3108        params[M4] = 0.31*params[M0]; (*See Carnall 1989,Table I,
        caption,probably fixed based on HF values*)
3109        params[P0] = 0;
3110        params[P4] = 0.5*params[P2];  (*See Carnall 1989,Table I,
        caption,probably fixed based on HF values*)
3111        params[P6] = 0.1*params[P2];  (*See Carnall 1989,Table I,
        caption,probably fixed based on HF values*)
3112        params[gs] = OptionValue["gs"];
3113        {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[
        F0], params[F2], params[F4], params[F6]];
3114        params[E0] = 0;
3115        If[
3116          Not[OptionValue["With Uncertainties"]],
```

```
3117        Return[params],
3118        (
3119          uncertain     = Association[Carnall["annotations"][Ln]];
3120          uncertainKeys = Keys[uncertain];
3121          uncertain     = If[# == "Not allowed to vary in fitting."
    || # == "Interpolated",
3122            0., #] & /@ uncertain;
3123          paramKeys = Keys[params];
3124          uncertainVals = Sort[Intersection[paramKeys, uncertainKeys
    ]] /. Association[uncertain];
3125          uncertainRules = MapThread[Rule, {Sort[uncertainKeys],
    uncertainVals}];
3126          Which[
3127            MemberQ[{"Ce", "Yb"}, Ln],
3128            (
3129              subsetL = {F0};
3130              subsetR = {0};
3131            ),
3132            True,
3133            (
3134            subsetL = {F0, M2, M4, P0, P4, P6, E0, E1, E2, E3};
3135            subsetR = {0, M0*0.65, M0*0.31, 0, P2*0.5, P2*0.1,
3136              0,
3137              Sqrt[(196 F2^2)/164025 + (49 F4^2)/88209 + (122500 F6
    ^2)/134165889],
3138              Sqrt[F2^2/4100625 + F4^2/10673289 + (30625 F6^2)
    /2743558264161],
3139              Sqrt[F2^2/18225 + (4 F4^2)/1185921 + (30625 F6^2)
    /1803785841]};
3140            )
3141          ];
3142          uncertainRules = Join[uncertainRules, MapThread[Rule, {
    subsetL,subsetR /. uncertainRules}]];
3143          uncertainRules = Association[uncertainRules];
3144          Which[
3145            Ln == "Eu",
3146            (
3147              uncertainRules[F4] = 12.121;
3148              uncertainRules[F6] = 15.872;
3149            ),
3150            Ln == "Gd",
3151            (
3152              uncertainRules[F4] = 12.07;
3153            ),
3154            Ln == "Tb",
3155            (
3156              uncertainRules[F4] = 41.006;
3157            )
3158          ];
3159          If[MemberQ[{"Eu","Gd","Tb"},Ln],
3160            (
3161              uncertainRules[E1] = Sqrt[(196 F2^2)/164025 + (49 F4^2)
    /88209 + (122500 F6^2)/134165889] /. uncertainRules;
3162              uncertainRules[E2] = Sqrt[F2^2/4100625 + F4^2/10673289
    + (30625 F6^2)/2743558264161] /. uncertainRules;
```

```
3163              uncertainRules[E3] = Sqrt[F2^2/18225 + (4 F4^2)/1185921
    + (30625 F6^2)/1803785841] /. uncertainRules;
3164             )
3165           ];
3166           uncertainKeys = First /@ Normal[uncertainRules];
3167           fullParams = Association[MapThread[Rule, {uncertainKeys,
    MapThread[Around, {uncertainKeys /. params, uncertainKeys /.
    uncertainRules}]}]];
3168           Return[Join[params, fullParams]]
3169         )
3170         ];
3171     )
3172   ];
3173
3174   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.";
3175   HoleElectronConjugation[params_] := Module[
3176     {newparams = params},
3177     (
3178       flipSignsOf = {ζ, T2, T3, T4, T6, T7, T8};
3179       flipSignsOf = Join[flipSignsOf, cfSymbols];
3180       flipped =
3181         Table[(flipper -> - newparams[flipper]),
3182         {flipper, flipSignsOf}
3183         ];
3184       nonflipped =
3185         Table[(flipper -> newparams[flipper]),
3186         {flipper, Complement[Keys[newparams], flipSignsOf]}
3187         ];
3188       flippedParams = Association[Join[nonflipped, flipped]];
3189       flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
3190       Return[flippedParams];
3191     )
3192   ];
3193
3194   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.
3195   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.
3196   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\".
3197   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.
```

```
3198  Tha ordered basis in which this eigenvector is to be interpreted is
        the one corresponding to BasisLSJMJ[numE].
3199  The function admits the following options:
3200  \"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.
3201  \"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.
3202  \"Zeroes\" (list) : A list with symbols assumed to be zero.
3203  ";
3204  Options[IonSolver] = {
3205    "Include Spin-Spin" -> True,
3206    "Overwrite Hamiltonian" -> False,
3207    "Zeroes" -> {}
3208  };
3209  IonSolver[numE_Integer, params0_Association, host_String:"Ln",
        OptionsPattern[]]:=Module[
3210    {ln, simplifier, simpleHam, numHam, eigensys,
3211    startTime, endTime, diagonalTime, params=params0, zeroSymbols},
3212    (
3213      ln = theLanthanides[[numE]];
3214
3215      (* This could be done when replacing values, but this produces
        smaller saved arrays. *)
3216      simplifier = (#-> 0) & /@ OptionValue["Zeroes"];
3217      simpleHam = SimplerSymbolicHamMatrix[numE,
3218        simplifier,
3219        "PrependToFilename" -> host,
3220        "Overwrite" -> OptionValue["Overwrite Hamiltonian"]
3221      ];
3222
3223      (* Note that we don't have to flip signs of parameters for fn
        beyond f7 since the matrix produced
3224      by SimplerSymbolicHamMatrix has already accounted for this. *)
3225
3226      (* Everything that is not given is set to zero *)
3227      params = ParamPad[params, "Print" -> True];
3228      PrintFun[params];
3229
3230      (* Enforce the override to the spin-spin contribution to the
        magnetic interactions *)
3231      params[\[Sigma]SS] = If[OptionValue["Include Spin-Spin"], 1,
        0];
3232
3233      (* Create the numeric hamiltonian *)
3234      numHam = ReplaceInSparseArray[simpleHam, params];
3235      Clear[simpleHam];
3236
3237      (* Eigensolver *)
3238      PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
3239      startTime = Now;
3240      eigensys  = Eigensystem[numHam];
```

```
3241        endTime    = Now;
3242        diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"
      ];
3243        PrintFun[">> Diagonalization took ", diagonalTime, " seconds."
      ];
3244        eigensys = Chop[eigensys];
3245        eigensys = Transpose[eigensys];
3246
3247        (* Shift the baseline energy *)
3248        eigensys = ShiftedLevels[eigensys];
3249        (* Sort according to energy *)
3250        eigensys = SortBy[eigensys, First];
3251        Return[eigensys];
3252      )
3253    ];
3254
3255    ShiftedLevels::usage = "ShiftedLevels[eigenSys] takes a list of
        levels of the form
3256    {{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.";
3257    ShiftedLevels[originalLevels_] := Module[
3258      {groundEnergy, shifted},
3259      (
3260        groundEnergy = Sort[originalLevels][[1,1]];
3261        shifted      = Map[{#[[1]] - groundEnergy, #[[2]]} &,
      originalLevels];
3262        Return[shifted];
3263      )
3264    ];
3265
3266    (* ############################################################ *)
3267    (* ################### Eigensystem analysis ################### *)
3268
3269    PrettySaundersSLJmJ::usage = "PrettySaundersSLJmJ[{SL, J, mJ}]
        produces a human-redeable symbol for the given basis vector {SL, J
        , mJ}."
3270    Options[PrettySaundersSLJmJ] = {"Representation" -> "Ket"};
3271    PrettySaundersSLJmJ[{SL_, J_, mJ_}, OptionsPattern[]] := (If[
3272      StringQ[SL],
3273      ({S, L} = FindSL[SL];
3274        L = StringTake[SL, {2, -1}];
3275        ),
3276      {S, L} = SL];
3277      pretty = RowBox[{AdjustmentBox[Style[2*S + 1, Smaller],
3278          BoxBaselineShift -> -1, BoxMargins -> 0],
3279          AdjustmentBox[PrintL[L], BoxMargins -> -0.2],
3280          AdjustmentBox[
3281          Style[{InputForm[J], mJ}, Small, FontTracking -> "Narrow"],
3282          BoxBaselineShift -> 1,
3283          BoxMargins -> {{0.7, 0}, {0.4, 0.4}}]}];
3284      pretty = DisplayForm[pretty];
3285      If[OptionValue["Representation"] == "Ket",
3286        pretty = Ket[pretty]
3287      ];
```

```
    Return[pretty];
  );

  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."
  BasisVecInRusselSaunders[basisVec_] := (
    {LSstring, Jval, mJval} = basisVec;
    Ket[PrettySaundersSLJmJ[basisVec]]
  );

  LSJMJTemplate =
    StringTemplate[
    "\!\(\*TemplateBox[{\nRowBox[{\"‘LS‘\", \",\", \nRowBox[{\"J\", \
\"=\", \"‘J‘\"}], \",\", \nRowBox[{\"mJ\", \"=\", \"‘mJ‘\"}]}]},\n\
\"Ket\"]\)"];

  BasisVecInLSJMJ::usage = "BasisVecInLSJMJ[basisVec] takes a basis
    vector in the format {{{LSstring, Jval}, mJval}, nucSpin} and
    returns a human-readable symbol for the corresponding LSJMJ term
    in the form |LS, J=..., mJ=...>."
  BasisVecInLSJMJ[basisVec_] := (
    {LSstring, Jval, mJval} = basisVec;
    LSJMJTemplate[<|
      "LS" -> LSstring,
      "J" -> ToString[Jval, InputForm],
      "mJ" -> ToString[mJval, InputForm]|>]
  );

  ParseStates::usage = "ParseStates[eigenSys, basis] takes a list of
    eigenstates in terms of their coefficients in the given basis and
    returns a list of the same states in terms of their energy, LSJMJ
    symbol, J, mJ, S, L, LSJ symbol, and LS symbol. eigenSys is a list
     of lists with two elements, in each list the first element is the
     energy and the second one the corresponding eigenvector. The LS
    symbol returned corresponds to the term with the largest
    coefficient in the given basis.";
  ParseStates[states_, basis_, OptionsPattern[]]:=Module[
    {parsedStates},
    (
      parsedStates = Table[(
        {energy, eigenVec}      = state;
        maxTermIndex            = Ordering[Abs[eigenVec]][[-1]];
        {LSstring, Jval, mJval} = basis[[maxTermIndex]];
        LSJsymbol               = Subscript[LSstring, {Jval, mJval}];
        LSJMJsymbol             = LSstring <> ToString[Jval,
    InputForm];
        {S, L}                  = FindSL[LSstring];
        {energy, LSstring, Jval, mJval, S, L, LSJsymbol, LSJMJsymbol}
      ),
      {state, states}
      ];
      Return[parsedStates];
    )
```

```
];

ParseStatesByNumBasisVecs::usage = "ParseStatesByNumBasisVecs[
  eigenSys, basis, numBasisVecs, roundTo] takes a list of
  eigenstates (given in eigenSys) in terms of their coefficients in
  the given basis and returns a list of the same states in terms of
  their energy and the coefficients at most numBasisVecs basis
  vectors. By default roundTo is 0.01 and this is the value used to
  round the amplitude coefficients. eigenSys is a list of lists with
   two elements, in each list the first element is the energy and
  the second one the corresponding eigenvector.
The option \"Coefficients\" can be used to specify whether the
  coefficients are given as \"Amplitudes\" or \"Probabilities\". The
   default is \"Amplitudes\".
";
Options[ParseStatesByNumBasisVecs] = {"Coefficients" -> "Amplitudes
  ", "Representation" -> "Ket"};
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}
       ),
```

```
3369          {state, eigensys}];
3370        Return[parsedStates]
3371      )
3372    ];
3373
3374    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.";
3375    FindThresholdPosition[list_, threshold_] := Module[
3376      {position},
3377      (
3378      position = Position[list, _?(# >= threshold &), 1, 1];
3379      thrPos = If[Length[position] > 0,
3380        position[[1, 1]],
3381        Length[list]];
3382      If[thrPos == 0,
3383        Return[1],
3384        Return[thrPos]]
3385      )
3386    ];
3387
3388    ParseStateByProbabilitySum[{energy_, eigenVec_}, probSum_, roundTo_
      :0.01, maxParts_:20] := Compile[
3389      {{energy, _Real, 0}, {eigenVec, _Complex, 1},
3390       {probSum, _Real, 0},{roundTo, _Real, 0},
3391       {maxParts, _Integer, 0}},
3392        Module[
3393          {numStates, state, amplitudes, probs, ordering,
3394          orderedProbs, truncationIndex, accProb, thresholdIndex,
      chosenIndices, majorComponents,
3395          majorAmplitudes, absMajorAmplitudes, notnullAmplitudes,
      majorRep},
3396          (
3397            numStates     = Length[eigenVec];
3398            (*Round them up*)
3399            amplitudes          = Round[eigenVec, roundTo];
3400            probs               = Round[Abs[eigenVec^2], roundTo];
3401            ordering            = Reverse[Ordering[probs]];
3402            (*Order the probabilities from high to low*)
3403            orderedProbs        = probs[[ordering]];
3404            (*To speed up Accumulate, assume that only as much as
      maxParts will be needed*)
3405            truncationIndex     = Min[maxParts, Length[orderedProbs]];
3406            orderedProbs        = orderedProbs[[;;truncationIndex]];
3407            (*Accumulate the probabilities*)
3408            accProb             = Accumulate[orderedProbs];
3409            (*Find the index of the first element in accProb that is
      greater than probSum*)
3410            thresholdIndex      = Min[Length[accProb],
      FindThresholdPosition[accProb, probSum]];
3411            (*Grab all the indicees up till that one*)
3412            chosenIndices       = ordering[[;; thresholdIndex]];
3413            (*Select the corresponding elements from the basis*)
```

```
3414            majorComponents    = basis[[chosenIndices]];
3415            (*Select the corresponding amplitudes*)
3416            majorAmplitudes    = amplitudes[[chosenIndices]];
3417            (*Take their absolute value*)
3418            absMajorAmplitudes = Abs[majorAmplitudes];
3419            (*Make sure that there are no effectively zero
    contributions*)
3420            notnullAmplitudes  = Flatten[Position[absMajorAmplitudes,
    x_ /; x != 0]];
3421            (* majorComponents    = PrettySaundersSLJmJ
    [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
3422            majorComponents    = PrettySaundersSLJmJ /@ majorComponents
    ;
3423            majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
3424            (*Make them into Kets*)
3425            majorComponents    = Ket /@ majorComponents[[
    notnullAmplitudes]];
3426            (*Multiply and add to build the final Ket*)
3427            majorRep           = majorAmplitudes . majorComponents;
3428            Return[{energy, majorRep}];
3429          )
3430        ],
3431      CompilationTarget -> "C",
3432      RuntimeAttributes -> {Listable},
3433      Parallelization -> True,
3434      RuntimeOptions -> "Speed"
3435    ];
3436
3437    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.";
3438    ParseStatesByProbabilitySum[eigensys_, basis_, probSum_, roundTo_ :
       0.01, maxParts_: 20]:=Module[
3439      {parsedByProb, numStates, state, energy, eigenVec, amplitudes,
      probs, ordering,
3440       orderedProbs, truncationIndex, accProb, thresholdIndex,
      chosenIndices, majorComponents,
3441       majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep
      },
3442       (
3443         numStates    = Length[eigensys];
3444         parsedByProb = Table[(
3445           state              = eigensys[[idx]];
3446           {energy, eigenVec} = state;
3447           (*Round them up*)
3448           amplitudes         = Round[eigenVec, roundTo];
3449           probs              = Round[Abs[eigenVec^2], roundTo];
3450           ordering           = Reverse[Ordering[probs]];
3451           (*Order the probabilities from high to low*)
3452           orderedProbs       = probs[[ordering]];
3453           (*To speed up Accumulate, assume that only as much as
      maxParts will be needed*)
3454           truncationIndex    = Min[maxParts, Length[orderedProbs]];
```

```
3455        orderedProbs       = orderedProbs[[;;truncationIndex]];
3456        (*Accumulate the probabilities*)
3457        accProb            = Accumulate[orderedProbs];
3458        (*Find the index of the first element in accProb that is
      greater than probSum*)
3459        thresholdIndex     = Min[Length[accProb],
      FindThresholdPosition[accProb, probSum]];
3460        (*Grab all the indicees up till that one*)
3461        chosenIndices      = ordering[[;; thresholdIndex]];
3462        (*Select the corresponding elements from the basis*)
3463        majorComponents    = basis[[chosenIndices]];
3464        (*Select the corresponding amplitudes*)
3465        majorAmplitudes    = amplitudes[[chosenIndices]];
3466        (*Take their absolute value*)
3467        absMajorAmplitudes = Abs[majorAmplitudes];
3468        (*Make sure that there are no effectively zero contributions
      *)
3469        notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_
      /; x != 0]];
3470        (* majorComponents    = PrettySaundersSLJmJ
      [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
3471        majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
3472        majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
3473        majorComponents    = majorComponents[[notnullAmplitudes]];
3474        (*Multiply and add to build the final Ket*)
3475        majorRep           = majorAmplitudes . majorComponents;
3476        {energy, majorRep}
3477        ), {idx, numStates}];
3478      Return[parsedByProb];
3479    )
3480  ];
3481
3482  (* ################## Eigensystem analysis ################## *)
3483  (* ########################################################## *)
3484
3485  (* ########################################################## *)
3486  (* ########################### Misc ######################### *)
3487
3488  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.";
3489  SymbToNum[expr_, replacementAssociation_]:= (
3490    includedKeys = Keys[replacementAssociation];
3491    (*If a key is not defined, make its value zero.*)
3492    fullAssociation = Table[(
3493      If[MemberQ[includedKeys, key],
3494        ToExpression[key]->replacementAssociation[key],
3495        ToExpression[key]->0
3496      ]
3497    ),
3498    {key, paramSymbols}];
3499    Return[expr/.fullAssociation];
3500  );
```

```
3501
3502  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.";
3503  SimpleConjugate[expr_] := expr /. Complex[a_, b_] :> a - I b;
3504
3505  ExportMZip::usage="ExportMZip[\"dest.[zip,m]\"] saves a compressed
      version of expr to the given destination.";
3506  ExportMZip[filename_, expr_]:=Module[
3507    {baseName, exportName, mImportName, zipImportName},
3508    (
3509      baseName    = FileBaseName[filename];
3510      exportName  = StringReplace[filename,".m"->".zip"];
3511      mImportName = StringReplace[exportName,".zip"->".m"];
3512      If[FileExistsQ[mImportName],
3513      (
3514        PrintTemporary[mImportName<>" exists already, deleting"];
3515        DeleteFile[mImportName];
3516        Pause[2];
3517      )
3518      ];
3519      Export[exportName, (baseName<>".m") -> expr];
3520    )
3521  ];
3522
3523  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";
3524  Options[ImportMZip]={"Leave Uncompressed" -> True};
3525  ImportMZip[filename_String, OptionsPattern[]]:=Module[
3526    {baseName, importKey, zipImportName, mImportName, imported},
3527    (
3528      baseName      = FileBaseName[filename];
3529      (*Function allows for the filename to be .m or .zip*)
3530      importKey     = baseName <> ".m";
3531      zipImportName = StringReplace[filename, ".m"->".zip"];
3532      mImportName   = StringReplace[zipImportName, ".zip"->".m"];
3533      If[FileExistsQ[mImportName],
3534      (
3535        PrintTemporary[".m version exists already, importing that
      instead ..."];
3536        Return[Import[mImportName]];
3537      )
3538      ];
3539      imported = Import[zipImportName, importKey];
3540      If[OptionValue["Leave Uncompressed"],
3541        Export[mImportName, imported]
3542      ];
3543      Return[imported]
```

```
3544      )
3545    ];
3546
3547    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.";
3548    ReplaceInSparseArray[sparseA_SparseArray, rules_]:=(
3549      SparseArray[Automatic,
3550        sparseA["Dimensions"],
3551        sparseA["Background"] /. rules,
3552        {
3553          1,
3554          {sparseA["RowPointers"], sparseA["ColumnIndices"]},
3555          sparseA["NonzeroValues"] /. rules
3556        }
3557        ]
3558    );
3559
3560    MapToSparseArray::usage = "MapToSparseArray[sparseArray, function]
          takes a sparse array and returns a sparse array after the function
           has been applied to it.";
3561    MapToSparseArray[sparseA_SparseArray, func_]:=Module[
3562      {nonZ, backg, mapped},
3563      (
3564        nonZ   = func /@ sparseA["NonzeroValues"];
3565        backg  = func[sparseA["Background"]];
3566        mapped = SparseArray[Automatic,
3567          sparseA["Dimensions"],
3568          backg,
3569          {
3570            1,
3571            {sparseA["RowPointers"], sparseA["ColumnIndices"]},
3572            nonZ
3573          }
3574        ];
3575        Return[mapped];
3576      )
3577    ];
3578
3579    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.";
3580    Options[ParseTeXLikeSymbol] = {"Form" -> "List"};
3581    ParseTeXLikeSymbol[bigString_, OptionsPattern[]] := Module[
3582      {form, mainSymbol, symbols},
3583      (
3584        form = OptionValue["Form"];
3585        (* parse greek *)
3586        symbols = Table[(
3587            str = StringReplace[string, {"\\alpha" -> "α",
```

```
3588            "\\beta" -> "β",
3589            "\\gamma" -> "γ",
3590            "\\psi" -> "\[Psi]"}];
3591          symbol = Which[
3592            StringContainsQ[str, "_"] && Not[StringContainsQ[str, "^"
    ]],
3593            (
3594              (*yes sub no sup*)
3595              mainSymbol = StringSplit[str, "_"][[1]];
3596              mainSymbol = ToExpression[mainSymbol];
3597
3598              subPart =
3599                StringCases[str,
3600                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
3601              Subscript[mainSymbol, subPart]
3602            ),
3603            Not[StringContainsQ[str, "_"]] && StringContainsQ[str, "^
    "],
3604            (
3605              (*no sub yes sup*)
3606              mainSymbol = StringSplit[str, "^"][[1]];
3607              mainSymbol = ToExpression[mainSymbol];
3608
3609              supPart =
3610                StringCases[str,
3611                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
3612              Superscript[mainSymbol, supPart]
3613            ),
3614            StringContainsQ[str, "_"] && StringContainsQ[str, "^"],
3615            (
3616              (*yes sub yes sup*)
3617              mainSymbol = StringSplit[str, "_"][[1]];
3618              mainSymbol = ToExpression[mainSymbol];
3619              {subPart, supPart} =
3620                StringCases[str, RegularExpression@"\\{(.*?)\\}" -> "
    $1"];
3621              Subsuperscript[mainSymbol, subPart, supPart]
3622            ),
3623            True,
3624            (
3625              (*no sup or sub*)
3626              str
3627            )
3628            ];
3629          symbol
3630          ),
3631        {string, StringSplit[bigString, " "]}
3632      ];
3633      Which[
3634        form == "Row",
3635        Return[Row[symbols]],
3636        form == "List",
3637        Return[symbols]
3638      ]
3639    )
```

```
];

(* ########################## 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};
          )}]]];
```

```
3740
3741  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.";
3742  Options[LabeledGrid]={
3743      ItemSize->Automatic,
3744      Alignment->Center,
3745      Frame->All,
3746      "Separator"->",",
3747      "Pivot"->""
3748  };
3749  LabeledGrid[data_,rowHeaders_,columnHeaders_,OptionsPattern[]]:=
        Module[
3750      {gridList=data,rowHeads=rowHeaders,colHeads=columnHeaders},
3751      (
3752        separator=OptionValue["Separator"];
3753        pivot=OptionValue["Pivot"];
3754        gridList=Table[
3755                Tooltip[
3756                  data[[rowIdx,colIdx]],
3757                  DisplayForm[
3758                    RowBox[{rowHeads[[rowIdx]],
3759                            separator,
3760                            colHeads[[colIdx]]}
3761                          ]
3762                          ]
3763                      ],
3764            {rowIdx,Dimensions[data][[1]]},
3765            {colIdx,Dimensions[data][[2]]}];
3766        gridList=Transpose[Prepend[gridList,colHeads]];
3767        rowHeads=Prepend[rowHeads,pivot];
3768        gridList=Prepend[gridList,rowHeads]//Transpose;
3769        Grid[gridList,
3770            Frame->OptionValue[Frame],
3771            Alignment->OptionValue[Alignment],
3772            Frame->OptionValue[Frame],
3773            ItemSize->OptionValue[ItemSize]
3774            ]
3775      )
3776  ];
3777
3778  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.";
3779  Options[HamiltonianForm]={"Separator"->","Pivot"->""}
3780  HamiltonianForm[hamMatrix_, basisLabels_List, OptionsPattern[]]:=(
3781      braLabels=DisplayForm[RowBox[{"\[LeftAngleBracket]",#,"\[
        RightBracketingBar]"}]]& /@ basisLabels;
```

```
3782        ketLabels=DisplayForm[RowBox[{"\[LeftBracketingBar]",#,"\[
    RightAngleBracket]"}]]& /@ basisLabels;
3783        LabeledGrid[hamMatrix,braLabels,ketLabels,"Separator"->
    OptionValue["Separator"],"Pivot"->OptionValue["Pivot"]]
3784        )
3785
3786    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.";
3787    Options[HamiltonianMatrixPlot] = Join[Options[MatrixPlot], {"Hover"
       -> True, "Overlay Values" -> True}];
3788    HamiltonianMatrixPlot[hamMatrix_, basisLabels_, opts :
    OptionsPattern[]] := (
3789     braLabels = DisplayForm[RowBox[{"\[LeftAngleBracket]", #, "\[
    RightBracketingBar]"}]] & /@ basisLabels;
3790     ketLabels = DisplayForm[Rotate[RowBox[{"\[LeftBracketingBar]", #,
     "\[RightAngleBracket]"}],\[Pi]/2]] & /@ basisLabels;
3791     ketLabelsUpright = DisplayForm[RowBox[{"\[LeftBracketingBar]", #,
     "\[RightAngleBracket]"}]] & /@ basisLabels;
3792     numRows = Length[hamMatrix];
3793     numCols = Length[hamMatrix[[1]]];
3794     epiThings = Which[
3795       And[OptionValue["Hover"], Not[OptionValue["Overlay Values"]]],
3796       Flatten[
3797         Table[
3798           Tooltip[
3799             {
3800               Transparent,
3801               Rectangle[
3802               {j - 1, numRows - i},
3803               {j - 1, numRows - i} + {1, 1}
3804               ]
3805             },
3806           Row[{braLabels[[i]],ketLabelsUpright[[j]],"=",hamMatrix[[i,
     j]]}]]
3807           ],
3808         {i, 1, numRows},
3809         {j, 1, numCols}
3810         ]
3811       ],
3812       And[OptionValue["Hover"], OptionValue["Overlay Values"]],
3813       Flatten[
3814         Table[
3815           Tooltip[
3816             {
3817               Transparent,
3818               Rectangle[
3819               {j - 1, numRows - i},
3820               {j - 1, numRows - i} + {1, 1}
3821               ]
3822             },
```

```
3823            DisplayForm[RowBox[{"\[LeftAngleBracket]", basisLabels[[i
    ]], "\[LeftBracketingBar]", basisLabels[[j]], "\[RightAngleBracket
    ]"}]]
3824            ],
3825          {i, numRows},
3826          {j, numCols}
3827          ]
3828        ],
3829        True,
3830        {}
3831        ];
3832      textOverlay = If[OptionValue["Overlay Values"],
3833        (
3834          Flatten[
3835          Table[
3836            Text[hamMatrix[[i, j]],
3837              {j - 1/2, numRows - i + 1/2}
3838            ],
3839          {i, 1, numRows},
3840          {j, 1, numCols}
3841          ]
3842          ]
3843        ),
3844        {}
3845        ];
3846      epiThings = Join[epiThings, textOverlay];
3847      MatrixPlot[hamMatrix,
3848        FrameTicks -> {
3849          {Transpose[{Range[Length[braLabels]], braLabels}], None},
3850          {None, Transpose[{Range[Length[ketLabels]], ketLabels}]}
3851          },
3852        Evaluate[FilterRules[{opts}, Options[MatrixPlot]]],
3853        Epilog -> epiThings
3854      ]
3855    );
3856
3857    (* ################## Some Plotting Routines ################## *)
3858    (* ############################################################ *)
3859
3860    (* ############################################################ *)
3861    (* ###################### Load Functions ###################### *)
3862
3863    LoadAll::usage="LoadAll[] executes most Load* functions.";
3864    LoadAll[]:=(
3865      LoadTermLabels[];
3866      LoadCFP[];
3867      LoadUk[];
3868      LoadV1k[];
3869      LoadT22[];
3870      LoadSOOandECSOLS[];
3871
3872      LoadElectrostatic[];
3873      LoadSpinOrbit[];
3874      LoadSOOandECSO[];
3875      LoadSpinSpin[];
```

```
3876      LoadThreeBody[];
3877      LoadChenDeltas[];
3878      LoadCarnall[];
3879    );
3880
3881    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.";
3882
3883    LoadTermLabels::usage="LoadTermLabels[] loads into the session the
          labels for the terms in the f^n configurations.";
3884    LoadTermLabels[]:= (
3885      If[ValueQ[fnTermLabels], Return[]];
3886      PrintTemporary["Loading data for state labels in the f^n
          configurations..."];
3887      fnTermsFname = FileNameJoin[{moduleDir, "data", "fnTerms.m"}];
3888
3889      If[!FileExistsQ[fnTermsFname],
3890        (PrintTemporary[">> fnTerms.m not found, generating ..."];
3891          fnTermLabels = ParseTermLabels["Export"->True];
3892        ),
3893        fnTermLabels = Import[fnTermsFname];
3894      ];
3895    );
3896
3897    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}";
3898
3899    LoadCarnall::usage="LoadCarnall[] loads data for trivalent
          lanthanides in LaF3 using the data from Bill Carnall's 1989 paper.
          ";
3900    LoadCarnall[]:=(
3901      If[ValueQ[Carnall], Return[]];
3902      carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
3903      If[!FileExistsQ[carnallFname],
3904        (PrintTemporary[">> Carnall.m not found, generating ..."];
3905          Carnall = ParseCarnall[];
3906        ),
3907        Carnall = Import[carnallFname];
3908      ];
3909    );
3910
3911    LoadChenDeltas::usage="LoadChenDeltas[] loads the differences noted
           by Chen.";
3912    LoadChenDeltas[]:=(
3913      If[ValueQ[chenDeltas], Return[]];
3914      PrintTemporary["Loading the association of discrepancies found by
           Chen ..."];
3915      chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas.m"
          }];
```

```
    If[!FileExistsQ[chenDeltasFname],
      (PrintTemporary[">> chenDeltas.m not found, generating ..."];
        chenDeltas = ParseChenDeltas[];
      ),
      chenDeltas = Import[chenDeltasFname];
    ];
  );

  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";
  Options[ParseChenDeltas] = {"Export" -> True};
  ParseChenDeltas[OptionsPattern[]]:=(
    chenDeltasRaw = Import[FileNameJoin[{moduleDir, "data", "the-chen
    -deltas-A.csv"}]];
    chenDeltasRaw = chenDeltasRaw[[2 ;;]];
    chenDeltas = <||>;
    chenDeltasA = <||>;
    Off[Power::infy];
    Do[
      ({right, wrong} = {chenDeltasRaw[[row]][[4 ;;]],
        chenDeltasRaw[[row + 1]][[4 ;;]]};
      key = chenDeltasRaw[[row]][[1 ;; 3]];
      repRule = (#[[1]] -> #[[2]]*#[[1]]) & /@
        Transpose[{{M0, M2, M4, P2, P4, P6}, right/wrong}];
      chenDeltasA[key] = <|"right" -> right, "wrong" -> wrong,
        "repRule" -> repRule|>;
      chenDeltasA[{key[[1]], key[[3]], key[[2]]}] = <|"right" ->
    right,
        "wrong" -> wrong, "repRule" -> repRule|>;
      ),
      {row, 1, Length[chenDeltasRaw], 2}];
    chenDeltas["A"] = chenDeltasA;

    chenDeltasRawB = Import[FileNameJoin[{moduleDir, "data", "the-
    chen-deltas-B.csv"}], "Text"];
    chenDeltasB = StringSplit[chenDeltasRawB, "\n"];
    chenDeltasB = StringSplit[#, ","] & /@ chenDeltasB;
    chenDeltasB = {ToExpression[StringTake[#[[1]], {2}]], #[[2]],
    #[[3]]} & /@ chenDeltasB;
    chenDeltas["B"] = chenDeltasB;
    On[Power::infy];
    If[OptionValue["Export"],
      (chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas
    .m"}];
      Export[chenDeltasFname, chenDeltas];
      )
      ];
    Return[chenDeltas];
  );

  ParseCarnall::usage="ParseCarnall[] parses the data found in ./data
    /Carnall.xls. If the option \"Export\" is set to True (True is the
     default), then the parsed data is saved to ./data/Carnall. This
```

```
    data is from the tables and appendices of Carnall et al (1989).";
Options[ParseCarnall] = {"Export" -> True};
ParseCarnall[OptionsPattern[]] := (
    ions         = {"Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho"
    ,"Er","Tm","Yb"};
    templates    = StringTemplate/@StringSplit["appendix:`ion`:
    Association appendix:`ion`:Calculated appendix:`ion`:RawTable
    appendix:`ion`:Headings"," "];

    (* How many unique eigenvalues, after removing Kramer's
    degeneracy *)
    fullSizes    = AssociationThread[ions, {7, 91, 182, 1001, 1001,
    3003, 1716, 3003, 1001, 1001, 182, 91, 7}];
    carnall      = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
    "}]][[2]];
    carnallErr   = Import[FileNameJoin[{moduleDir,"data","Carnall.xls
    "}]][[3]];

    elementNames = carnall[[1]][[2;;]];
    carnall      = carnall[[2;;]];
    carnallErr   = carnallErr[[2;;]];
    carnall      = Transpose[carnall];
    carnallErr   = Transpose[carnallErr];
    paramNames   = ToExpression/@carnall[[1]][[1;;]];
    carnall      = carnall[[2;;]];
    carnallErr   = carnallErr[[2;;]];
    carnallData  = Table[(
                    data         = carnall[[i]];
                    data         = (#[[1]]->#[[2]])&/@Select[
    Transpose[{paramNames,data}],#[[2]]!=""&];
                    elementNames[[i]]->data
                    ),
                    {i,1,13}
                    ];
    carnallData  = Association[carnallData];
    carnallNotes = Table[(
                    data        = carnallErr[[i]];
                    elementName = elementNames[[i]];
                    dataFun     = (
                        #[[1]] -> If[#[[2]]=="[]",
                        "Not allowed to vary in fitting.",
                        If[#[[2]]=="[R]",
                            "Ratio constrained by: " <> <|"Eu"->"F4/
    F2=0.713; F6/F2=0.512",
                                "Gd"->"F4/F2=0.710]",
                                "Tb"->"F4/F2=0.707"|>[elementName],
                            If[#[[2]]=="i",
                                "Interpolated",
                                #[[2]]
                            ]
                        ]
                        ]) &;
                    data = dataFun /@ Select[Transpose[{paramNames,
    data}],#[[2]]!=""&];
                    elementName->data
```

```
                    ),
                {i,1,13}
                ];
    carnallNotes  = Association[carnallNotes];

    annotatedData = Table[
                If[NumberQ[#[[1]]],Tooltip[#[[1]],#[[2]]],""] & /@ Transpose[{paramNames/.carnallData[element],
                  paramNames/.carnallNotes[element]
                    }],
                {element,elementNames}
                ];
    annotatedData = Transpose[annotatedData];

    Carnall = <|"data"       -> carnallData,
        "annotations"        -> carnallNotes,
        "paramSymbols"       -> paramNames,
        "elementNames"       -> elementNames,
        "rawData"            -> carnall,
        "rawAnnotations"     -> carnallErr,
        "includedTableIons" -> ions,
        "annnotatedData"     -> annotatedData
    |>;

    Do[(
        carnallData   = Import[FileNameJoin[{moduleDir,"data","Carnall.xls"}]][[sheetIdx]];
        headers       = carnallData[[1]];
        calcIndex     = Position[headers,"Calc (1/cm)"][[1,1]];
        headers       = headers[[2;;]];
        carnallLabels = carnallData[[1]];
        carnallData   = carnallData[[2;;]];
        carnallTerms  = DeleteDuplicates[First/@carnallData];
        parsedData    = Table[(
                          rows = Select[carnallData,#[[1]]==term&];
                          rows = #[[2;;]]&/@rows;
                          rows = Transpose[rows];
                          rows = Transpose[{headers,rows}];
                          rows = Association[(#[[1]]->#[[2]])&/@rows];
                          term->rows
                          ),
                        {term,carnallTerms}
                        ];
        carnallAssoc         = Association[parsedData];
        carnallCalcEnergies = #[[calcIndex]]&/@carnallData;
        carnallCalcEnergies = If[NumberQ[#],#,Missing[]]&/@carnallCalcEnergies;
        ion                  = ions[[sheetIdx-3]];
        carnallCalcEnergies = PadRight[carnallCalcEnergies, fullSizes[ion], Missing[]];
        keys                 = #[<|"ion"->ion|>]&/@templates;
        Carnall[keys[[1]]]   = carnallAssoc;
        Carnall[keys[[2]]]   = carnallCalcEnergies;
        Carnall[keys[[3]]]   = carnallData;
```

```
4055          Carnall[keys[[4]]]  = headers;
4056          ),
4057      {sheetIdx,4,16}
4058      ];
4059
4060      goodions = Select[ions,#!="Pm"&];
4061      expData  = Select[Transpose[Carnall["appendix:"<>#<>":RawTable"
          ]][[1+Position[Carnall["appendix:"<>#<>":Headings"],"Exp (1/cm)"
          ][[1,1]]]],NumberQ]&/@goodions;
4062      Carnall["All Experimental Data"]=AssociationThread[goodions,
          expData];
4063      If[OptionValue["Export"],
4064        (
4065          carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"
          }];
4066          Print["Exporting to "<>carnallFname];
4067          Export[carnallFname, Carnall];
4068        )
4069        ];
4070      Return[Carnall];
4071    );
4072
4073    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";
4074
4075    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[].";
4076
4077    LoadCFP::usage="LoadCFP[] loads CFP, CFPAssoc, and CFPTable into
        the session.";
4078    LoadCFP[]:=(
4079      If[And[ValueQ[CFP], ValueQ[CFPTable], ValueQ[CFPAssoc]],Return
        []];
4080
4081      PrintTemporary["Loading CFPtable ..."];
4082      CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"}];
4083      If[!FileExistsQ[CFPTablefname],
4084        (PrintTemporary[">> CFPTable.m not found, generating ..."];
4085          CFPTable = GenerateCFPTable["Export"->True];
4086        ),
4087        CFPTable = Import[CFPTablefname];
4088      ];
4089
4090      PrintTemporary["Loading CFPs.m ..."];
4091      CFPfname = FileNameJoin[{moduleDir, "data", "CFPs.m"}];
4092      If[!FileExistsQ[CFPfname],
4093        (PrintTemporary[">> CFPs.m not found, generating ..."];
4094          CFP = GenerateCFP["Export"->True];
```

```
4095        ),
4096        CFP = Import[CFPfname];
4097      ];
4098
4099      PrintTemporary["Loading CFPAssoc.m ..."];
4100      CFPAfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"}];
4101      If[!FileExistsQ[CFPAfname],
4102        (PrintTemporary[">> CFPAssoc.m not found, generating ..."];
4103          CFPAssoc = GenerateCFPAssoc["Export"->True];
4104        ),
4105        CFPAssoc = Import[CFPAfname];
4106      ];
4107    );
4108
4109    ReducedUkTable::usage = "ReducedUkTable[{n, 1 = 3, SL, SpLp, k}]
       provides reduced matrix elements of the spherical tensor operator
       Uk. See TASS section 11-9 \"Unit Tensor Operators\". Loaded using
       LoadUk[].";
4110
4111    LoadUk::usage="LoadUk[] loads into session the reduced matrix
       elements for unit tensor operators.";
4112    LoadUk[]:=(
4113      If[ValueQ[ReducedUkTable], Return[]];
4114      PrintTemporary["Loading the association of reduced matrix
       elements for unit tensor operators ..."];
4115      ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
       ReducedUkTable.m"}];
4116      If[!FileExistsQ[ReducedUkTableFname],
4117        (PrintTemporary[">> ReducedUkTable.m not found, generating ..."
       ];
4118          ReducedUkTable = GenerateReducedUkTable[7];
4119        ),
4120        ReducedUkTable = Import[ReducedUkTableFname];
4121      ];
4122    );
4123
4124    ElectrostaticTable::usage = "ElectrostaticTable[{n, SL, SpLp}]
       provides the calculated result of Electrostatic[{n, SL, SpLp}].
       Load using LoadElectrostatic[].";
4125
4126    LoadElectrostatic::usage="LoadElectrostatic[] loads the reduced
       matrix elements for the electrostatic interaction.";
4127    LoadElectrostatic[]:=(
4128      If[ValueQ[ElectrostaticTable], Return[]];
4129      PrintTemporary["Loading the association of matrix elements for
       the electrostatic interaction ..."];
4130      ElectrostaticTablefname = FileNameJoin[{moduleDir, "data", "
       ElectrostaticTable.m"}];
4131      If[!FileExistsQ[ElectrostaticTablefname],
4132        (PrintTemporary[">> ElectrostaticTable.m not found, generating
       ..."];
4133          ElectrostaticTable = GenerateElectrostaticTable[7];
4134        ),
4135        ElectrostaticTable = Import[ElectrostaticTablefname];
4136      ];
```

```
4137  );
4138
4139  LoadV1k::usage="LoadV1k[] loads into session the matrix elements of
        V1k.";
4140  LoadV1k[]:=(
4141    If[ValueQ[ReducedV1kTable], Return[]];
4142    PrintTemporary["Loading the association of matrix elements for
      V1k ..."];
4143    ReducedV1kTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedV1kTable.m"}];
4144    If[!FileExistsQ[ReducedV1kTableFname],
4145      (PrintTemporary[">> ReducedV1kTable.m not found, generating ...
      "];
4146        ReducedV1kTable = GenerateReducedV1kTable[7];
4147      ),
4148      ReducedV1kTable = Import[ReducedV1kTableFname];
4149    ]
4150  );
4151
4152  LoadSpinOrbit::usage="LoadSpinOrbit[] loads into session the matrix
       elements of the spin-orbit interaction.";
4153  LoadSpinOrbit[]:=(
4154    If[ValueQ[SpinOrbitTable], Return[]];
4155    PrintTemporary["Loading the association of matrix elements for
      spin-orbit ..."];
4156    SpinOrbitTableFname = FileNameJoin[{moduleDir, "data", "
      SpinOrbitTable.m"}];
4157    If[!FileExistsQ[SpinOrbitTableFname],
4158      (PrintTemporary[">> SpinOrbitTable.m not found, generating ..."
      ];
4159        SpinOrbitTable = GenerateSpinOrbitTable[7, "Export" -> True];
4160      ),
4161      SpinOrbitTable = Import[SpinOrbitTableFname];
4162    ]
4163  );
4164
4165  LoadSOOandECSOLS::usage="LoadSOOandECSOLS[] loads into session the
      LS reduced matrix elements of the SOO-ECSO interaction.";
4166  LoadSOOandECSOLS[]:=(
4167    If[ValueQ[SOOandECSOLSTable], Return[]];
4168    PrintTemporary["Loading the association of LS reduced matrix
      elements for SOO-ECSO ..."];
4169    SOOandECSOLSTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedSOOandECSOLSTable.m"}];
4170    If[!FileExistsQ[SOOandECSOLSTableFname],
4171      (PrintTemporary[">> ReducedSOOandECSOLSTable.m not found,
      generating ..."];
4172        SOOandECSOLSTable = GenerateSOOandECSOLSTable[7];
4173      ),
4174      SOOandECSOLSTable = Import[SOOandECSOLSTableFname];
4175    ];
4176  );
4177
4178  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 *)

```

```
    finalHam = compHam @@ (complementaryArgs /. bestParams);
    {eigenEnergies, eigenVecs} = Eigensystem[finalHam];
    ordering = Ordering[eigenEnergies];
    eigenEnergies = eigenEnergies - Min[eigenEnergies];
    states = Transpose[{eigenEnergies, eigenVecs}];
    states = states[[ordering]];
    coarseStates = ParseStates[states, basis];

    (* The eigenvectors need to be simplified in order to compare
  labels to labels *)
    coarseStates = {#[[1]], #[[-1]]} & /@ coarseStates;
    missingLevels = Length[coarseStates] - Length[expData];
    (* The energies are in the first element of the tuples. *)
    energyDiffFun = (Abs[#1[[1]] - #2[[1]]]) &;
    (* matching disregarding labels to get overall scale for
  scaling differences in labels *)
    energyFlow = FlowMatching[coarseStates,
            expData,
            "notMatched" -> missingLevels,
            "CostFun" -> energyDiffFun];
    energyPairs = {#[[1]][[1]], #[[2]][[1]]} & /@ energyFlow[[1]];
    energyRms = Sqrt[Total[(Abs[#[[2]] - #[[1]]])^2 & /@
  energyPairs] / Length[energyPairs]];
    (* matching using both labels and energies *)
    energyAndLabelFun = (Abs[#1[[1]] - #2[[1]]] + EditDistance
  [#1[[2]],#2[[2]]]) &;
    energyAndLabelFlow = FlowMatching[coarseStates,
          expData,
          "notMatched" -> (Length[coarseStates] - Length[expData]),
          "CostFun" -> energyAndLabelFun];
    totalAvgCost = Total[energyAndLabelFun @@ # & /@
  energyAndLabelFlow[[1]]] / Length[energyAndLabelFlow[[1]]];

    compa = (Flatten /@ energyAndLabelFlow[[1]]);
    compa = Join[
        #,
        {
          #[[2]] == #[[4]],
          If[NumberQ[#[[1]]],
            Round[#[[1]] - #[[3]],1],
            ""
          ],
        #[[5]] - #[[3]],
        Which[
          Round[Abs[#[[1]] - #[[3]]]] < Round[Abs[#[[5]] -
  #[[3]]]],
            "Better",
          Round[Abs[#[[1]] - #[[3]]]] == Round[Abs[#[[5]] -
  #[[3]]]],
            "Equal",
          True,
            "Worse"
        ]
        }
    ] & /@ 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
```

```
    .
588   \"MaxIterations\": determines the maximum number of iterations used
       by NMinimize.
589
590   \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
       3.
591   \"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.
592   \"PrintFun\": the function used to print progress messages, the
       default is PrintTemporary.
593   \"SlackChannel\": name of the Slack channel to which to dump
       progress messaages, the default is None which disables this option
        entirely.
594   \"ProgressView\": whether or not a progress window will be opened
       to show the progress of the solver, the default is True.
595
596   \"ReturnHashFileNameAndExit\": if True then the function returns
       the name of the file with the solutions and exits, the default is
       False.
597   \"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 {}.
598   \"DE:CrossProbability\": the cross probability used by the
       Differential Evolution method, the default is 0.5.
599   \"DE:ScalingFactor\": the scaling factor used by the Differential
       Evolution method, the default is 0.6.
600   \"DE:SearchPoints\": the number of search points used by the
       Differential Evolution method, the default is Automatic.
601
602   \"MagneticSimplifier\": a list of replacement rules to simplify the
        Marvin and pesudo-magnetic paramters.
603   \"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.
604   \"SymmetrySimplifier\": a list of replacements rules to simplify
       the crystal field.
605   \"OtherSimplifier\": an additiona list of replacement rules that
       are applied to the Hamiltonian before computing with it.
606   \"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.
607
608   \"FreeIonSymbols\": a list with the symbols to be included in the
       intermediate coupling basis.
609   \"AppendToLogFile\": an association appended to the log file under
       the key \"Appendix\".
610   ";
611 Options[TruncationFit]={
612   "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 f^",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"],
```

```
1461        StepMonitor :> (
1462          steps       += 1;
1463          currentRMS = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues
    @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j,
    presentDataIndices}];
1464          currentRMS = Sqrt[currentRMS / degressOfFreedom];
1465          paramSols  = AddToList[paramSols, constrainedProblemVarsList,
     maxHistory];
1466          rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
1467          )
1468        ];
1469      endTime = Now;
1470      timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
1471      PrintFun["Solution found in ", timeTaken, "s"];
1472
1473      solVec = constrainedProblemVars /. sol[[-1]];
1474      indepSolVec = indepVars /. sol[[-1]];
1475      If[addShift,
1476        \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
1477        \[Epsilon]Best = 0
1478      ];
1479      fullSolVec = standardValues;
1480      fullSolVec[[problemVarsPositions]] = solVec;
1481      PrintFun["Calculating the numerical Hamiltonian corresponding to
    the solution ..."];
1482      fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
1483      PrintFun["Calculating energies and eigenvectors ..."];
1484      {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
1485      states = Transpose[{eigenEnergies, eigenVectors}];
1486      states = SortBy[states, First];
1487      eigenEnergies = First /@ states;
1488      PrintFun["Shifting energies to make ground state zero of energy
    ..."];
1489      eigenEnergies = eigenEnergies - eigenEnergies[[1]];
1490      PrintFun["Calculating the linear approximant to each eigenvalue
    ..."];
1491      allVarsVec = Transpose[{allVars}];
1492      p0 = Transpose[{fullSolVec}];
1493      linMat = {};
1494      If[addShift,
1495        tail = -2,
1496        tail = -1];
1497      Do[
1498        (
1499          aVarPosition = Position[allVars, aVar][[1, 1]];
1500          isolationValues = ConstantArray[0, Length[allVars]];
1501          isolationValues[[aVarPosition]] = 1;
1502          dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
    constraints]];
1503          Do[
1504            isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
1505            {dVar, dependentVars}
1506          ];
1507          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 *)
\[Mu]0  = 4 \[Pi]*10^-7;             (* magnetic permeability in vacuum in SI *)
hPlanck = 6.62607015*10^-34;         (* Planck's constant in SI *)
\[Mu]B  = 9.2740100783*10^-24;       (* Bohr magneton in SI *)
me      = 9.1093837015*10^-31;       (* electron mass in SI *)
cLight  = 2.99792458*10^8;           (* speed of light in SI *)
eCharge = 1.602176634*10^-19;        (* elementary charge in SI *)
αFine = 1/137.036;             (* fine structure constant in SI *)
bohrRadius   = 5.29177210903*10^-11; (* Bohr radius in SI *)

(* 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 *)
eVtoKayser    = 8065.54429;   (* 1 eV = 8065.54429 cm^-1 *)
KaysertoeV    = 1/eVtoKayser; (* 1 cm^-1 = 1/8065.54429 eV *)
TeslaToKayser = 2 * \[Mu]B / hPlanck / cLight / 100;

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,
  (
```

```
144       R = ((wavelength - 510)/(580 - 510))^gamma;
145       G = 1.0;
146       B = 0.0;
147       ),
148     580 <= wavelength <= 645,
149     (
150       R = 1.0;
151       G = (-(wavelength - 645)/(645 - 580))^gamma;
152       B = 0.0;
153       ),
154     645 <= wavelength <= 750,
155     (
156       attenuation = 0.3 + 0.7*(750 - wavelength)/(750 - 645);
157       R = (1.0*attenuation)^gamma;
158       G = 0.0;
159       B = 0.0;
160       ),
161     True,
162     (
163       R = 0;
164       G = 0;
165       B = 0;
166       )];
167     Return[RGBColor[R, G, B]]
168     )
169
170   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.";
171   FuzzyRectangle[xCenter_, width_, ymin_, height_, color_, intensity_
      :1] := Module[
172     {intenseColor, nocolor, ymax, polys},
173   (
174     nocolor = Directive[Opacity[0], color];
175     ymax = ymin + height;
176     intenseColor = Directive[Opacity[intensity], color];
177     polys = {
178       Polygon[{
179         {xCenter - width/2, ymin},
180         {xCenter, ymin},
181         {xCenter, ymax},
182         {xCenter - width/2, ymax}},
183         VertexColors -> {
184           nocolor,
185           intenseColor,
186           intenseColor,
187           nocolor,
188           nocolor}],
189       Polygon[{
190         {xCenter, ymin},
191         {xCenter + width/2, ymin},
```

```
192         {xCenter + width/2, ymax},
193         {xCenter, ymax}},
194         VertexColors -> {
195           intenseColor,
196           nocolor,
197           nocolor,
198           intenseColor,
199           intenseColor}]
200       };
201     Return[polys]
202     );
203   ]
204
205   Options[SpectrumPlot] = Options[Graphics];
206   Options[SpectrumPlot] = Join[Options[SpectrumPlot], {"Intensities"
      -> {},"Tooltips" -> True, "Comments" -> {}, "SpectrumFunction" ->
      WaveToRGB}];
207   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 with 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.
208   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.
209   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.
210   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"];
```

```
    {minWave, maxWave} = MinMax[lines];
    height     = (maxWave - minWave)/widthToHeightAspect;
    fuzzyLines = Which[
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] == 0,
        FuzzyRectangle[#, lineWidth, 0, height, colorFun[#]] &/@ lines,
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] == 0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1]] &, {lines, lineWidth}],
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] > 0,
        MapThread[FuzzyRectangle[#1, lineWidth, 0, height, colorFun[#1], #2] &, {lines, OptionValue["Intensities"]}],
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] > 0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1], #3] &, {lines, lineWidth, OptionValue["Intensities"]}]
    ];
    comments = Which[
      Length[OptionValue["Comments"]] > 0,
        MapThread[StringJoin[ToString[#1]<>" nm","\n",ToString[#2]]&,
        {lines, OptionValue["Comments"]}],
      Length[OptionValue["Comments"]] == 0,
        ToString[#] <>" nm" & /@ lines,
      True,
      {}
    ];
    If[OptionValue["Tooltips"],
      fuzzyLines = MapThread[Tooltip[#1, #2] &, {fuzzyLines, comments}];
    ];
    graphicsOpts = FilterRules[{opts}, Options[Graphics]];
    Graphics[fuzzyLines,
        graphicsOpts,
        Background -> Black,
        PlotRange -> {{minWave, maxWave}, {0, height}}]
    )
  ];

End[];

EndPackage[];
```

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;

Begin["`Private`"];

  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},
```

```
53  (
54    accumulatedBlockSize = Accumulate[blockSizes];
55    If[accumulatedBlockSize[[-1]]-index<0,
56      Print["Index out of bounds"];
57      Abort[]
58    ];
59    blockIndex    = Flatten[Position[accumulatedBlockSize-index,n_ /;
       n>=0]][[1]];
60    blockSubIndex = Mod[index-accumulatedBlockSize[[blockIndex]],
      blockSizes[[blockIndex]],1];
61    Return[{blockIndex,blockSubIndex}]
62  )
63  ];
64
65  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.";
66  TruncateBlockArray[blockArray_,truncationIndices_,blockWidths_]:=
      Module[
67  {truncatedArray,blockCol,blockRow,blockSubCol,blockSubRow},(
68  truncatedArray = Table[
69    {blockCol,blockSubCol} = BlockAndIndex[blockWidths,fullColIndex];
70    {blockRow,blockSubRow} = BlockAndIndex[blockWidths,fullRowIndex];
71    blockArray[[blockRow,blockCol]][[blockSubRow,blockSubCol]],
72  {fullColIndex,truncationIndices},
73  {fullRowIndex,truncationIndices}
74  ];
75  Return[truncatedArray]
76  )
77  ];
78
79  BlockArrayDimensionsArray::usage="BlockArrayDimensionsArray[
      blockArray] returns the array of block sizes in a given blocked
      array.";
80  BlockArrayDimensionsArray[blockArray_]:=(
81    Map[Dimensions,blockArray,{2}]
82  );
83
84  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.";
85  ArrayBlocker[anArray_,blockSizes_]:=Module[{rowStart,colStart,
      colEnd,numBlocks,blockedArray,blockSize,rowEnd,aBlock,idxRow,
      idxCol},(
86    rowStart   = 1;
87    colStart   = 1;
88    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;
```

```
134       {roundedX, roundedDx} = If[sigFigs <= 0,
135         {0., N@RoundToSignificantFigures[dx, 1]},
136         N[
137         {
138           RoundToSignificantFigures[x, xExpo - dxExpo + 1],
139           RoundToSignificantFigures[dx, 1]}
140         ]
141         ];
142       returning = If[
143         OptionValue["SetPrecision"],
144         {SetPrecision[roundedX, Max[1, sigFigs]],
145         SetPrecision[roundedDx, 1]},
146         {roundedX, roundedDx}
147         ];
148       Return[returning]
149       )
150     ];
151
152   RoundToSignificantFigures::usage =
153     "RoundToSignificantFigures[x, sigFigs] rounds x so that it only
      has \
154   sigFigs significant figures.";
155   RoundToSignificantFigures[x_, sigFigs_] :=
156     Sign[x]*N[FromDigits[RealDigits[x, 10, sigFigs]]];
157
158   RobustMissingQ[expr_] := (FreeQ[expr, _Missing] === False);
159
160   TextBasedProgressBar[progress_, totalIterations_, prefix_:""] :=
      Module[
161       {progMessage},
162       progMessage = ToString[progress] <> "/" <> ToString[
      totalIterations];
163       If[progress < totalIterations,
164           WriteString["stdout", StringJoin[prefix, progMessage, "\r"
      ]],
165           WriteString["stdout", StringJoin[prefix, progMessage, "\n"
      ]]
166       ];
167   ];
168
169   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.";
170   FirstOrderPerturbation[eigenVectors_,
171     perMatrix_] := (Diagonal[
172     eigenVectors . perMatrix . Transpose[eigenVectors]])
173
174   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[#]] &];
    Return[{flatbasisLS, flatbasisNums}]
    )
  ];

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
  }.";
RecoverBasis[{flatbasisLS_, flatbasisNums_}] := Module[{recBasis},
  (
  recBasis = {{{#[[1]], #[[2]]}, #[[3]]}, #[[4]]} & /@ (Flatten /@
      Transpose[{flatbasisLS,
        Partition[Round[2*#]/2 & /@ flatbasisNums, 3]}]);
  Return[recBasis];
  )
  ]

```

```
ExtractSymbolNames[expr_Hold] := Module[
  {strSymbols},
  strSymbols = ToString[expr, InputForm];
  StringCases[strSymbols, RegularExpression["\\w+"]][[2 ;;]]
  ]

ExportToH5::usage =
  "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.
   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.";
Options[ExportToH5] = {"Overwrite" -> True};
ExportToH5[fname_String, symbols_Hold, OptionsPattern[]] := (
  If[And[FileExistsQ[fname], OptionValue["Overwrite"]],
  (
    Print["File already exists, overwriting ..."];
    DeleteFile[fname];
    )
  ];
  symbolNames = ExtractSymbolNames[symbols];
  Do[(Print[symbolName];
    Export[fname, ToExpression[symbolName], {"Datasets", symbolName
  },
    OverwriteTarget -> "Append"]
    ), {symbolName, symbolNames}]
  )

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.";
Options[GreedyMatching] = {
    "alistLabels" -> {},
    "blistLabels" -> {}};
GreedyMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
  aValues = aValues0,
  bValues = bValues0,
  bValuesOriginal = bValues0,
  bestLabels, bestMatches,
  bestLabel, aElement, givenLabels,
  aLabels, aLabel,
  diffs, minDiff,
  bLabels,
```

```
    minDiffPosition, bestMatch},
    (
    aLabels     = OptionValue["alistLabels"];
    bLabels     = OptionValue["blistLabels"];
    bestMatches = {};
    bestLabels  = {};
    givenLabels = (Length[aLabels] > 0);
    Do[
      (
      aElement         = aValues[[idx]];
      diffs            = Abs[bValues - aElement];
      minDiff          = Min[diffs];
      minDiffPosition = Position[diffs, minDiff][[1, 1]];
      bestMatch        = bValues[[minDiffPosition]];
      bestMatches      = Append[bestMatches, {aElement, bestMatch}];
      If[givenLabels,
        (
        aLabel     = aLabels[[idx]];
        bestLabel  = bLabels[[minDiffPosition]];
        bestLabels = Append[bestLabels, {aLabel, bestLabel}];
        bLabels    = Drop[bLabels, {minDiffPosition}];
        )
        ];
      bValues = Drop[bValues, {minDiffPosition}];
      If[Length[bValues] == 0, Break[]];
      ),
      {idx, 1, Length[aValues]}
      ];
    pairedIndices = MapIndexed[{#2[[1]], Position[bValuesOriginal,
    #1[[2]]][[1, 1]]} &, bestMatches];
    If[givenLabels,
      Return[{bestMatches, bestLabels, pairedIndices}],
      Return[{bestMatches, pairedIndices}]
      ]
    )
    ]

  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.";
  Options[StochasticMatching] = {"alistLabels" -> {},
    "blistLabels" -> {}};
  StochasticMatching[aValues0_, bValues0_, numShuffles_ : 200,
    OptionsPattern[]] := Module[{
    aValues = aValues0,
    bValues = bValues0,
```

```
288     matchingLabels, ranger, matches, noShuff, bestMatch, highestCost,
         lowestCost, dev, sorter, bestValues,
289     pairedIndices, bestLabels, matchedIndices, shuffler
290     },
291     (
292     matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
293     ranger = Range[1, Length[aValues]];
294     matches = If[Not[matchingLabels], (
295         Table[(
296           shuffler = If[i == 1, ranger, RandomSample[ranger]];
297           {bestValues, matchedIndices} =
298           GreedyMatching[aValues[[shuffler]], bValues];
299           cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
300           {cost, {bestValues, matchedIndices}}
301           ), {i, 1, numShuffles}]
302       ),
303       Table[(
304         shuffler = If[i == 1, ranger, RandomSample[ranger]];
305         {bestValues, bestLabels, matchedIndices} =
306           GreedyMatching[aValues[[shuffler]], bValues,
307           "alistLabels" -> OptionValue["alistLabels"][[shuffler]],
308           "blistLabels" -> OptionValue["blistLabels"]];
309         cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
310         {cost, {bestValues, bestLabels, matchedIndices}}
311         ), {i, 1, numShuffles}]
312       ];
313     noShuff = matches[[1, 1]];
314     matches = SortBy[matches, First];
315     bestMatch = matches[[1, 2]];
316     highestCost = matches[[-1, 1]];
317     lowestCost = matches[[1, 1]];
318     dev = StandardDeviation[First /@ matches];
319     Print[lowestCost, " <-> ", highestCost, " | \[Sigma]=", dev,
320       " | N=", numShuffles, " | null=", noShuff];
321     If[matchingLabels,
322       (
323       {bestValues, bestLabels, matchedIndices} = bestMatch;
324       sorter = Ordering[First /@ bestValues];
325       bestValues = bestValues[[sorter]];
326       bestLabels = bestLabels[[sorter]];
327       pairedIndices =
328         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
329         bestValues];
330       Return[{bestValues, bestLabels, pairedIndices}]
331       ),
332       (
333       {bestValues, matchedIndices} = bestMatch;
334       sorter = Ordering[First /@ bestValues];
335       bestValues = bestValues[[sorter]];
336       pairedIndices =
337         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
338         bestValues];
339       Return[{bestValues, pairedIndices}]
340       )
341       ];
```

```
342     )
343     ]
344
345   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.";
346   Options[FlowMatching] = {"alistLabels" -> {}, "blistLabels" -> {},
       "notMatched" -> 0, "CostFun"-> (Abs[#1-#2] &)};
347   FlowMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
348     aValues = aValues0, bValues = bValues0, edgesSourceToA,
       capacitySourceToA, nA, nB,
349     costSourceToA, midLayer, midLayerEdges, midCapacities,
350     midCosts, edgesBtoSink, capacityBtoSink, costBtoSink,
351     allCapacities, allCosts, allEdges, graph,
352     flow, bestValues, bestLabels, cFun,
353     aLabels, bLabels, pairedIndices, matchingLabels},
354     (
355     matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
356     aLabels = OptionValue["alistLabels"];
357     bLabels = OptionValue["blistLabels"];
358     cFun    = OptionValue["CostFun"];
359     nA      = Length[aValues];
360     nB      = Length[bValues];
361     (*Build up the edges costs and capacities*)
362     (*From source to the nodes representing the values of the first \
363   list*)
364     edgesSourceToA    = ("source" \[DirectedEdge] {"A", #}) & /@
       Range[1, nA];
365     capacitySourceToA = ConstantArray[1, nA];
366     costSourceToA     = ConstantArray[0, nA];
367
368     (*From all the elements of A to all the elements of B*)
369     midLayer = Table[{{"A", i} \[DirectedEdge] ({"B", j}), 1, cFun[
       aValues[[i]], bValues[[j]]]}, {i, 1, nA}, {j, 1, nB}];
370     midLayer = Flatten[midLayer, 1];
371     {midLayerEdges, midCapacities, midCosts} = Transpose[midLayer];
372
373     (*From the elements of B to the sink*)
374     edgesBtoSink    = ({"B", #} \[DirectedEdge] "sink") & /@ Range[1,
        nB];
375     capacityBtoSink = ConstantArray[1, nB];
```

```
376     costBtoSink     = ConstantArray[0, nB];
377
378     (*Put it all together*)
379     allCapacities = Join[capacitySourceToA, midCapacities,
        capacityBtoSink];
380     allCosts      = Join[costSourceToA, midCosts, costBtoSink];
381     allEdges      = Join[edgesSourceToA, midLayerEdges, edgesBtoSink
        ];
382     graph         = Graph[allEdges, EdgeCapacity -> allCapacities,
383       EdgeCost -> allCosts];
384
385     (*Solve it*)
386     flow          = FindMinimumCostFlow[graph, "source", "sink", nA -
         OptionValue["notMatched"], "OptimumFlowData"];
387     (*Collect the pairs of matched indices*)
388     pairedIndices = Select[flow["EdgeList"], And[Not[#[[1]] === "
        source"], Not[#[[2]] === "sink"]] &];
389     pairedIndices = {#[[1, 2]], #[[2, 2]]} & /@ pairedIndices;
390     (*Collect the pairs of matched values*)
391     bestValues    = {aValues[[#[[1]]]], bValues[[#[[2]]]]} & /@
        pairedIndices;
392     (*Account for having been given labels*)
393     If[matchingLabels,
394       (
395       bestLabels = {aLabels[[#[[1]]]], bLabels[[#[[2]]]]} & /@
        pairedIndices;
396       Return[{bestValues, bestLabels, pairedIndices}]
397       ),
398       (
399       Return[{bestValues, pairedIndices}]
400       )
401       ];
402     )
403     ]
404
405   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.";
406   HelperNotebook[nbName_:"OUT"] :=
407   Module[{screenDims, screenWidth, screenHeight, nbWidth, leftMargin,
408     PrintToOutputNb}, (
409     screenDims =
410       SystemInformation["Devices", "ScreenInformation"][[1, 2, 2]];
411     screenWidth = screenDims[[1, 2]];
412     screenHeight = screenDims[[2, 2]];
413     nbWidth = Round[screenWidth/3];
414     leftMargin = screenWidth - nbWidth;
415     outputNb = CreateDocument[{}, WindowTitle -> nbName,
416       WindowMargins -> {{leftMargin, Automatic}, {Automatic,
417           Automatic}}, WindowSize -> {nbWidth, screenHeight}];
418     PrintToOutputNb[text_] :=
419       (
420           SelectionMove[outputNb, After, Notebook];
421           NotebookWrite[outputNb, Cell[BoxData[ToBoxes[text]], "
```

```
    Output"]];
422       );
423     Return[PrintToOutputNb]
424     )
425     ]
426
427   GetModificationDate::usage="GetModificationDate[fname] returns the
      modification date of the given file.";
428   GetModificationDate[theFileName_] := FileDate[theFileName, "
      Modification"];
429
430   (*Helper function to convert Mathematica expressions to standard
      form*)
431   StandardFormExpression[expr0_] := Module[{expr=expr0}, ToString[
      expr, InputForm]];
432
433   (*Helper function to translate to Python/SymPy expressions*)
434   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."
      ;
435   ToPythonSymPyExpression[expr0_] := Module[{standardForm, expr=expr0
      },
436     standardForm = StandardFormExpression[expr];
437     StringReplace[standardForm, {
438       "Power[" -> "Pow(",
439       "Sqrt[" -> "sqrt(",
440       "[" -> "(",
441       "]" -> ")",
442       "\\" -> "",
443       "I" -> "1j",
444       (*Remove special Mathematica backslashes*)
445       "/" -> "/" (*Ensure division is represented with a slash*)}]];
446
447   ToPythonSparseFunction[sparseArray_SparseArray, funName_] :=
448     Module[{data, rowPointers, columnIndices, dimensions, pyCode,
      vars,
449       varList, dataPyList,
450       colIndicesPyList},(*Extract unique symbolic variables from the
      \
451   SparseArray*)
452     vars = Union[Cases[Normal[sparseArray], _Symbol, Infinity]];
453     varList = StringRiffle[ToString /@ vars, ", "];
454     (*varList=ToPythonSymPyExpression/@varList;*)
455     (*Convert data to SymPy compatible strings*)
456     dataPyList =
457       StringRiffle[
458       ToPythonSymPyExpression /@ Normal[sparseArray["NonzeroValues"
      ]],
459       ", "];
460     colIndicesPyList =
461       StringRiffle[
462       ToPythonSymPyExpression /@ (Flatten[
463           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,
      StringTake[termN, {1, 2}],
      termN
      ],
      {termN, termNames}
    ];

    (*Load the parameters from Carnall*)
    PrintFun["> Loading the fit parameters from Carnall ..."];
    params = LoadParameters[ln, "Free Ion" -> False];
    If[numE>7,
      (
        PrintFun["> Conjugating the parameters accounting for the
    hole-particle equivalence ..."];
        params = HoleElectronConjugation[params];
        params[t2Switch] = 0;
      ),
      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];
118     PrintFun[params];
119     numHam = ReplaceInSparseArray[simpleHam, params];
120     If[Not[OptionValue["Sparse"]],
121         numHam = Normal[numHam]
122     ];
123     PrintFun["> Calculating the SLJ basis ..."];
124     basis = BasisLSJMJ[numE];
125
126     (* Eigensolver *)
127     PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
128     startTime = Now;
129     eigensys  = Eigensystem[numHam];
130     endTime   = Now;
131     diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"];
132     PrintFun[">> Diagonalization took ", diagonalTime, " seconds."];
133     eigensys = Chop[eigensys];
134     eigensys = Transpose[eigensys];
```

```
135
136     (* Shift the baseline energy *)
137     eigensys = ShiftedLevels[eigensys];
138     (* Sort according to energy *)
139     eigensys = SortBy[eigensys, First];
140     (* Grab just the energies *)
141     eigenEnergies = First /@ eigensys;
142
143     (* Energies are doubly degenerate in the case of odd number of
        electrons, keep only one *)
144     If[And[OddQ[numE], OptionValue["Remove Kramers"]],
145         (
146         PrintFun["> Since there's an odd number of electrons energies
         come in pairs, taking just one for each pair ..."];
147         eigenEnergies = eigenEnergies[[;; ;; 2]];
148         )
149     ];
150
151     (* Compare against the data quoted by Bill Carnall *)
152     PrintFun["> Comparing against the data from Carnall ..."];
153     mainKey            = StringTemplate["appendix:`Ln`:Association"
        ][<|"Ln" -> ln|>];
154     lnData             = Carnall[mainKey];
155     carnalKeys         = lnData // Keys;
156     repetitions        = Length[lnData[#]["Calc (1/cm)"]] & /@
        carnalKeys;
157     carnallAssignments = First /@ Carnall["appendix:" <> ln <> ":
        RawTable"];
158     carnallAssignments = Select[carnallAssignments, Not[# === ""] &];
159     carnalKey          = StringTemplate["appendix:`Ln`:Calculated"
        ][<|"Ln" -> ln|>];
160     carnallEnergies    = Carnall[carnalKey];
161
162     If[And[OddQ[numE], Not[OptionValue["Remove Kramers"]]],
163     (
164       PrintFun[">> The number of eigenstates and the number of quoted
         states don't match, removing the last state ..."];
165       carnallAssignments = Duplicator[carnallAssignments];
166       carnallEnergies    = Duplicator[carnallEnergies];
167     )
168     ];
169
170     (* For the difference take as many energies as quoted by Bill *)
171     eigenEnergies = eigenEnergies + carnallEnergies[[1]];
172     diffs = Sort[eigenEnergies][[;; Length[carnallEnergies]]] -
        carnallEnergies;
173     (* Remove the differences where the appendix tables have elided
        values*)
174     rmsDifference = Sqrt[Mean[(Select[diffs, FreeQ[#, Missing[]] &])
        ^2]];
175     titleTemplate = StringTemplate[
176       "Energy Level Diagram of \!\(\*SuperscriptBox[\(`ion`\),
        \(\(3\)\(+\)\)]\)"];
177     title = titleTemplate[<|"ion" -> ln|>];
178     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]",
        Count[assignmentMatches, "\[Checkmark]"]}, {"X",
        Count[assignmentMatches, "X"]}};
      labelComparison = (AssignmentMatchFunc /@ Transpose[{
    carnallAssignments, simplerStateLabels[[;; Length[
    carnallAssignments]]]}]);
      labelComparison = PadRight[labelComparison, Length[
    simplerStateLabels], "-"];

      statesTable = Grid[Prepend[{Round[#[[1]]], #[[2]]} & /@
          truncatedStates[[;;Min[Length[eigensys],maxStatesInTable
    ]]], {"Energy/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
          "\[Psi]"}], Frame -> All, Spacings -> {2, 2},
          FrameStyle -> Blue,
          Dividers -> {{False, True, False}, {True, True}}];
      DefaultIfMissing[expr_]:= If[FreeQ[expr, Missing[]], expr,"NA"
    ];
      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]] <> "\"}]];"
```

```
            ),"Input"
          ]
        ];
        NotebookWrite[nb,
          Cell[(
            "Manipulate[First[MinimalBy[truncatedStates, Abs[First[#] - energy] &]], {energy,0}]"
            ),"Input"]
        ];
        (* Move the cursor to the top of the notebook *)
        SelectionMove[nb, Before, Notebook];
        Export[exportFname,
          {rmsDifference, carnallEnergies, eigenEnergies, ln, carnallAssignments, simplerStateLabels, eigensys, basis, truncatedStates}
        ];
        tinyexportFname = FileNameJoin[
          {workDir, OptionValue["OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> " - tiny.m"}
        ];
        tinyExport = <|"ln"->ln,
                       "carnallEnergies"->carnallEnergies,
                       "rmsDifference"-> rmsDifference,
                       "eigenEnergies"-> eigenEnergies,
                       "carnallAssignments"-> carnallAssignments,
                       "simplerStateLabels" -> simplerStateLabels|>;
        Export[tinyexportFname, tinyExport];
      )
      ];
      If[OptionValue["NotebookSave"],
        (
          nbFname = FileNameJoin[{workDir,OptionValue["OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".nb"}];
          PrintFun[">> Saving notebook to ", nbFname, " ..."];
          NotebookSave[nb, nbFname];
        )
      ];
    )
    ];

    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>*)
```

```
361     \[Psi]i:idx,     (* index of the wavefuction |i>*)
362     \[Psi]f:idx,     (* index of the wavefuction |j>*)
363     Ei/K,            (* energy of the initial state in K *)
364     Ef/K,            (* energy of the final state in K *)
365     \[Lambda]/nm,    (* transition wavelength in nm *)
366     \[CapitalDelta]\[Lambda]/nm, (* uncertainty in the transition
        wavelength in nm *)
367     \[Tau]/s,        (* radiative lifetime in s *)
368     AMD/s^-1         (* magnetic dipole transition rate in s^-1 *)
369
370 The raw data file contains the following keys:
371     - Line Strength, (* Line strength array *)
372     - AMD, (* Magnetic dipole transition rates in 1/s *)
373     - fMD, (* Oscillator strengths from ground to excited states *)
374     - Radiative lifetimes, (* Radiative lifetimes in s *)
375     - Transition Energies / K, (* Transition energies in K *)
376     - Transition Wavelengths in nm. (* Transition wavelengths in nm
        *)
377
378 The CSV file contains the same information as the tabular file.
379
380 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.
381
382 The function takes the following options:
383     - \"Make Notebook\" -> True or False. If True, a notebook with a
        Manipulate is created. Default is True.
384     - \"Print Function\" -> PrintTemporary or Print. The function
        used to print the progress of the calculation. Default is
        PrintTemporary.
385     - \"Host\" -> \"LaF3\". The host material. Default is LaF3.
386     - \"Wavelength Range\" -> {50,2000}. The range of wavelengths in
        nm for the Manipulate object in the created notebook. Default is
        {50,2000}.
387
388 The function returns an association containing the following keys:
        Line Strength, AMD, fMD, Radiative lifetimes, Transition Energies
        / K, Transition Wavelengths in nm.";
389 Options[MagneticDipoleTransitions] = {
390         "Make Notebook" -> True,
391         "Close Notebook" -> True,
392         "Print Function" -> PrintTemporary,
393         "Host" -> "LaF3",
394         "Wavelength Range" -> {50,2000}};
395 MagneticDipoleTransitions[numE_Integer, OptionsPattern[]]:= (
396   host           = OptionValue["Host"];
397   \[Lambda]Range = OptionValue["Wavelength Range"];
398   PrintFun       = OptionValue["Print Function"];
399   {\[Lambda]min, \[Lambda]max} = OptionValue["Wavelength Range"];
400
401   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"};
402   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];
```

```
447 roundedEnergies = Round[eigenEnergies, 1.];
448 simpleFromTo    = Outer[{#1,#2}&, simplerStateLabels,
      simplerStateLabels];
449 fromTo          = Outer[{#1,#2}&, Range[numEigenvecs], Range[
      numEigenvecs]];
450 energyPairs     = Outer[{#1,#2}&, roundedEnergies,
      roundedEnergies];
451 allTransitions  = {simpleFromTo,
452     fromTo,
453     energyPairs,
454     magIon["Transition Wavelengths in nm"],
455     \[Lambda]uncertainty,
456     magIon["AMD"],
457     magIon["Radiative lifetimes"]
458 };
459 allTransitions = (Flatten/@Transpose[Flatten[#,1]&/@allTransitions
      ]);
460 allTransitions = Select[allTransitions, #[[3]]!=#[[4]]&];
461 allTransitions = Select[allTransitions, #[[10]]>0&];
462 allTransitions = Transpose[allTransitions];
463
464 (*round things up*)
465 PrintFun["Rounding wavelengths according to estimated uncertainties
      ..."];
466 {roundedWaves,roundedDeltas} = Transpose[MapThread[
      RoundValueWithUncertainty,{allTransitions[[7]],allTransitions
      [[8]]}]];
467 allTransitions[[7]]          = roundedWaves;
468 allTransitions[[8]]          = roundedDeltas;
469
470 PrintFun["Rounding lifetimes and transition rates to three
      significant figures ..."];
471 allTransitions[[9]]          = RoundToSignificantFigures[#,3]&/@(
      allTransitions[[9]]);
472 allTransitions[[10]]         = RoundToSignificantFigures[#,3]&/@(
      allTransitions[[10]]);
473 finalTable                   = Transpose[allTransitions];
474 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.

[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.

[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.

[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.