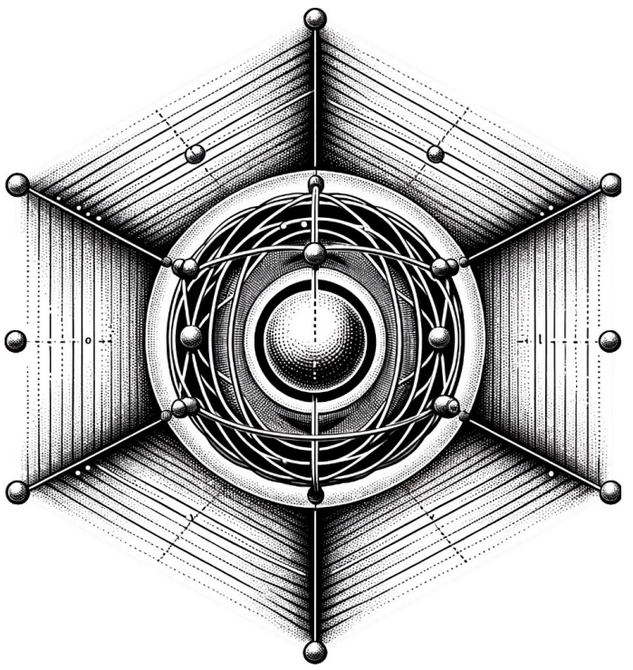


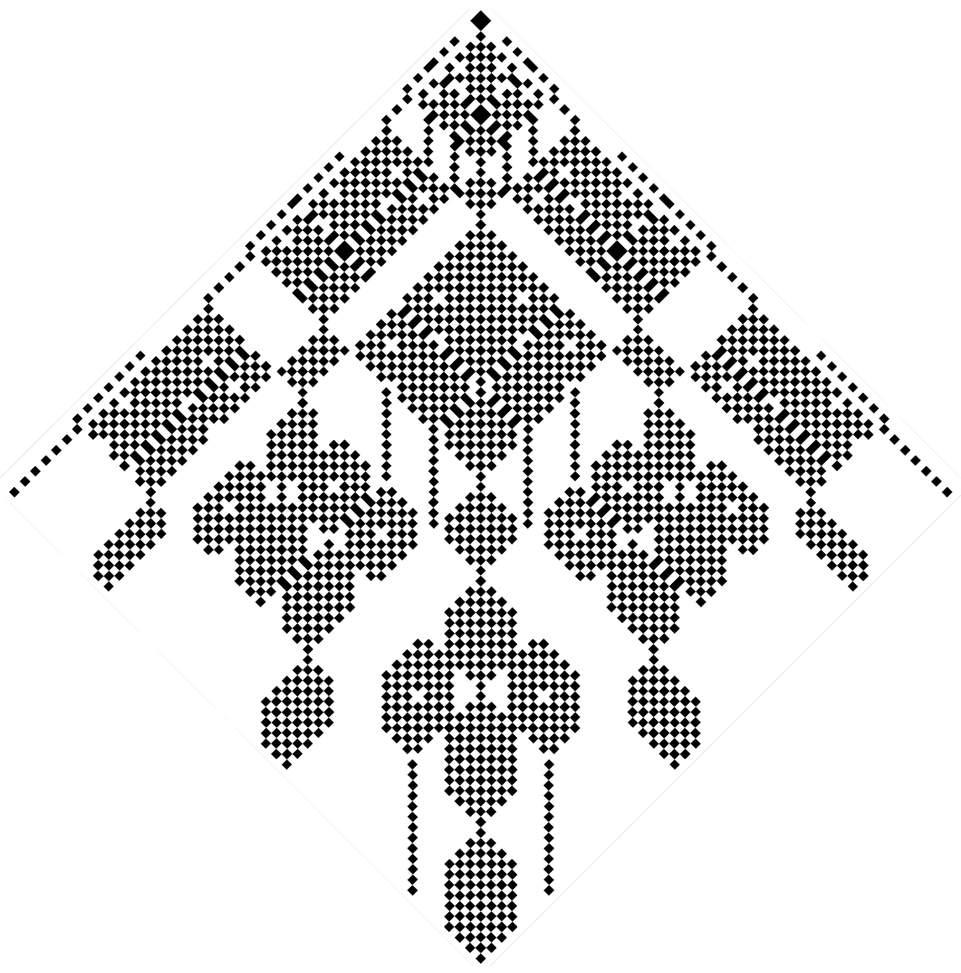
`qlanth`

doc version $|\alpha\rangle^{(11)}$

Juan David Lizarazo Ferro
& Christopher Dodson

Under the advisory of Dr. Rashid Zia

qlanth may be downloaded
here



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 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.

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

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

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

This document has 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 calculating "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. **Section 7** explains the details of fitting the Hamiltonian to experimental data. **Sections 7 and 8** list notebooks and additional data included in **qlanth**. **Section 9** has a brief comment on units. **Sections 9 and 10** include a summary of notation and definitions used throughout this document. Finally, **section 12** contains a printout of the code included in **qlanth**.

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

Contents

1 LS coupling **1**
- 1.1 $|LSJM_J\rangle$ states . . . 1
- 1.2 More quantun numbers . . . 5
  - 1.2.1 Seniority $\boldsymbol{\nu}$ . . . 5
  - 1.2.2 $\boldsymbol{\mathcal{U}}$ and $\boldsymbol{\mathcal{W}}$ . . . 5
- 1.3 $|LSJ\rangle$ levels . . . 5

2 The coefficients of fractional parentage **11**

3 The JJ' block structure **13**

4 The effective Hamiltonian **15**
- 4.1 $\hat{\mathcal{H}}_{\mathrm{k}}$: kinetic energy . . . 17
- 4.2 $\hat{\mathcal{H}}_{\mathrm{e:sn}}$: the central field potential . . . 17
- 4.3 $\hat{\mathcal{H}}_{\mathrm{e:e}}$: e:e repulsion . . . 18
- 4.4 $\hat{\mathcal{H}}_{\mathrm{s:o}}$: spin-orbit . . . 20
- 4.5 $\hat{\mathcal{H}}_{\mathcal{SO}(3)}, \hat{\mathcal{H}}_{\mathrm{G}_2}, \hat{\mathcal{H}}_{\mathcal{SO}(7)}$: electrostatic configuration interaction . . . 21
- 4.6 $\hat{\mathcal{H}}_{\mathrm{s:s\text{-}s:oo}}$: *spin-spin* and *spin-other-orbit* . . . 22
- 4.7 $\hat{\mathcal{H}}_{\mathrm{ecs:o}}$: *electrostatically-correlated-spin-orbit* . . . 26
- 4.8 $\hat{\mathcal{H}}_3$: three-body effective operators . . . 34
- 4.9 $\hat{\mathcal{H}}_{\mathrm{cf}}$: crystal-field . . . 39
- 4.10 $\hat{\mu}$ and $\hat{\mathcal{H}}_{\mathrm{Z}}$: the magnetic dipole operator and the Zeeman term . . . 42
- 4.11 Going beyond f^7 . . . 45

5 Transitions **45**
- 5.1 State description . . . 45
  - 5.1.1 Magnetic dipole transitions . . . 45
- 5.2 Level description . . . 48
  - 5.2.1 Forced electric dipole transitions . . . 48
  - 5.2.2 Magnetic dipole transitions . . . 48

6 Data fitting **51**

7 Accompanying notebooks **65**

8 Additional data **66**
- 8.1 Carnall et al data on Ln:LaF_3 . . . 66
- 8.2 `sparsefn.py` . . . 68

9 Units **69**

10 Notation **70**

11 Definitions **70**

12 code **71**
- 12.1 qlanth.m . . . 71
- 12.2 fittings.m . . . 150
- 12.3 qonstants.m . . . 180
- 12.4 qplotter.m . . . 181
- 12.5 misc.m . . . 186
- 12.6 qalculations.m . . . 194

1 LS coupling

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

qlanth uses LS coupling for its calculations. In LS coupling all the orbital angular momenta are added to form the total orbital angular momentum L, all the spin angular momenta are added to form the total spin angular momentum S, and finally these two angular momenta are then added together to form the total angular momentum J. The exclusion principle is taken into account in limiting the possible LS terms, and requires no further restrictions. Finally this total angular momentum J is complemented with the quantum number[1] M_J describing the proyection of J along the z-axis.

It is worthwhile remembering here the spectroscopic hierarchy of descriptive elements:

terms correspond to $|LS\rangle$ (also noted as ${}^{2S+1}\mathrm{L}$),
levels correspond to $|LSJ\rangle$ (also noted as ${}^{2S+1}\mathrm{L}_J$),
states correspond to $|LSJM_J\rangle$ (also noted as ${}^{2S+1}\mathrm{L}_{J,M_J}$).

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

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

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

1.1 $|LSJM_J\rangle$ states

The basis vectors of the $|LSJM_J\rangle$ basis are common eigenvectors of the operators $\hat{L}^2$, $\hat{S}^2$, $\hat{J}^2$, and $\hat{J}_z$. The are formed starting from the allowed LS terms in a given configuration, and are then completed with attendant J and M_J quantum numbers. The LS terms allowed in each configuration $\underline{\mathrm{f}}^n$ are obtained from tables that originate from the original work by Nielson and Koster [NK63]. In **qlanth** these are parsed from the file `B1F_ALL.TXT` which is part of the doctoral research of Dobromir Velkov (under the advisory of Brian Judd) [Vel00] in which he calculated anew the coefficients of fractional parentage.

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

The following are all the LS terms in the $\underline{\mathrm{f}}^n$ configurations. In the notation used the superscript index before the letter notes the spin multiplicity $2S+1$, the roman letter 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 (sadly, we prepend it, rather than append it).

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

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

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

---

^{2}F

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

---

^{2}F

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

---

^{1}S

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

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

In addition to LS the $|LSJM_J\rangle$ basis states are also specified by the total angular momentum J (which may go from $|L-S|$ to $|L+S|$). Then for each J there are $2J+1$ projections on the z-axis. For example, the ordered $|LSJM_J\rangle$ basis for $\underline{f}^2$ is 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 states)

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

$(J=1)$
(3 states)

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

$(J = 2)$
(15 states)

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

$(J = 3)$
(7 states)

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

$(J = 4)$
(27 states)

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

$(J = 5)$
(11 states)

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

$(J = 6)$
(26 states)

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

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

In **qlanth** the ordered basis used for a given $\underline{\mathrm{f}}^n$ is provided by `BasisLSJMJ`.

```
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]
  )
];
```

1.2 More quantun numbers

Besides using an integer which solves the problem of discriminating between degenerate LS terms by enumerating them, it is also possible to add more useful labels that reflect additional symmetries that the f-electron wavefunctions find in the groups $\mathcal{SO}(7)$ and $\mathcal{G}_2$.

1.2.1 Seniority $\boldsymbol{\nu}$

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 $\underline{\mathrm{f}}^{\bar{n}}$ in which this term appeared. For example $\underline{\mathrm{f}}^5$ contains six degenerate ^2G terms. The first time this term appeared was in $\underline{\mathrm{f}}^3$, where it had a degeneracy of 2. The 2 degenerate terms in $\underline{\mathrm{f}}^3$ would then both have a seniority of $\boldsymbol{\nu} = 3$ since they first appeared in $\underline{\mathrm{f}}^3$. In consequence two of the six degenerate terms in $\underline{\mathrm{f}}^5$ would have the same degeneracy those two in $\underline{\mathrm{f}}^3$, and are therefore linked to those previous two. The four remaining ones, are considered to be *born* in $\underline{\mathrm{f}}^5$, and therefore have a seniority $\boldsymbol{\nu} = 5$.

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

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 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 equally adequate, albeit being somewhat less parsimonious than what the quasi-spin view that seniority can provide. As such **qlanth** does not use the seniority numbers that are associated with each LS term.

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

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

In **qlanth** the $\mathcal{W}$ and $\mathcal{U}$ are used in order to determine the matrix elements of the $\mathcal{C}(\mathcal{SO}(7))$ and $\mathcal{C}(\mathcal{G}_2)$ Casimir operators.

1.3 $|LSJ\rangle$ levels

When the Hamiltonian only includes spherically symmetric terms (or what is the same, when the crystal field is neglected) then the M_J quantum numbers in the $|LSJM_J\rangle$ basis states are redundant. This permits a simplified description in terms of $|LSJ\rangle$ levels. The following are the different $^{2S+1}\mathrm{L}_J$ levels that span the eigenvectors that result from diagonalizing the Hamiltonian in the level description.

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

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

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

$\underline{\mathrm{f}}^4$ (107 LSJ levels)
$^5\mathrm{D}_0$, $^3\mathrm{P}_0$, $^3\mathrm{P}_0$, $^3\mathrm{P}_0$, $^1\mathrm{S}_0$, $^1\mathrm{S}_0$, $^5\mathrm{D}_1$, $^5\mathrm{F}_1$, $^3\mathrm{P}_1$, $^3\mathrm{P}_1$, $^3\mathrm{P}_1$, $^3\mathrm{D}_1$, $^3\mathrm{D}_1$, $^5\mathrm{S}_2$, $^5\mathrm{D}_2$, $^5\mathrm{F}_2$, $^5\mathrm{G}_2$, $^3\mathrm{P}_2$, $^3\mathrm{P}_2$, $^3\mathrm{P}_2$, $^3\mathrm{D}_2$, $^3\mathrm{D}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^3\mathrm{F}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^1\mathrm{D}_2$, $^5\mathrm{D}_3$, $^5\mathrm{F}_3$, $^5\mathrm{G}_3$, $^3\mathrm{D}_3$, $^3\mathrm{D}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{F}_3$, $^3\mathrm{G}_3$, $^3\mathrm{G}_3$, $^3\mathrm{G}_3$, $^1\mathrm{F}_3$, $^5\mathrm{D}_4$, $^5\mathrm{F}_4$, $^5\mathrm{G}_4$, $^5\mathrm{I}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{F}_4$, $^3\mathrm{G}_4$, $^3\mathrm{G}_4$, $^3\mathrm{G}_4$, $^3\mathrm{H}_4$, $^3\mathrm{H}_4$, $^3\mathrm{H}_4$,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\underline{f}^{12}$ (13 LSJ levels)

$^{3}P_{0}$, $^{1}S_{0}$, $^{3}P_{1}$, $^{3}P_{2}$, $^{3}F_{2}$, $^{1}D_{2}$, $^{3}F_{3}$, $^{3}F_{4}$, $^{3}H_{4}$, $^{1}G_{4}$, $^{3}H_{5}$, $^{3}H_{6}$, $^{1}I_{6}$

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

The level picture is a much more frugal description of the eigenstates. Not only the number of basis elements 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. One needs, however to use adequate degeneracy factors in the relevant calculations.

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

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

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

```
LevelSimplerSymbolicHamMatrix::usage = "LevelSimplerSymbolicHamMatrix
    [numE] is a variation of HamMatrixAssembly that returns the
    diagonal JJ Hamiltonian blocks applying a simplifier and with
    simplifications adequate for the level description. The keys of
    the given association correspond to the different values of J that
     are possible for f^numE, the values are sparse array that are
    meant to be interpreted in the basis provided by BasisLSJ.
The option \"Simplifier\" is a list of symbols that are set to zero.
    At a minimum this has to include the crystal field parameters. By
    default this includes everything except the Slater parameters Fk
    and the spin orbit coupling ζ.
The option \"Export\" controls whether the resulting association is
    saved to disk, the default is True and the resulting file is saved
     to the ./hams/ folder. A hash is appended to the filename that
    corresponds to the simplifier used in the resulting expression. If
     the option \"Overwrite\" is set to False then these files may be
    used to quickly retrieve a previously computed case. The file is
    saved both in .m and .mx format.
The option \"PrependToFilename\" can be used to append a string to
    the filename to which the function may export to.
The option \"Return\" can be used to choose whether the function
    returns the matrix or not.
The option \"Overwrite\" can be used to overwrite the file if it
    already exists.";
Options[LevelSimplerSymbolicHamMatrix] = {
  "Export" -> True,
  "PrependToFilename" -> "",
  "Overwrite" -> False,
  "Return" -> True,
  "Simplifier" -> Join[
    {F0,\[Sigma]SS},
    cfSymbols,
    TSymbols,
    casimirSymbols,
    pseudoMagneticSymbols,
    marvinSymbols,
    DeleteCases[magneticSymbols,ζ]
```

```
  ]
};
LevelSimplerSymbolicHamMatrix[numE_Integer, OptionsPattern[]] :=
    Module[
  {thisHamAssoc, Js, fname,
  fnamemx, hash, simplifier},
  (
    simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
    hash       = Hash[simplifier];
    If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
    If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
    If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
    If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
    If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
    fname    = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".m"}];
    fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".mx"}];
    If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue["
    Overwrite"]],
    (
      If[OptionValue["Return"],
      (
        Which[FileExistsQ[fnamemx],
        (
          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 description in terms of levels. To this aim the following functions in **qlanth** are instrumental: `EigenLever`, `FreeHam`, `ListRepeater`, and `ListLever`.

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

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

```
    numHam     = N /@ Map[ReplaceInSparseArray[#, params]&, simpleHam
  ];
  Clear[simpleHam];
  (* Eigensolver *)
  PrintFun["> Diagonalizing the numerical Hamiltonian within each
  separate J-subspace ..."];
  startTime    = Now;
  eigensys     = Eigensystem /@ numHam;
  endTime      = Now;
  diagonalTime = QuantityMagnitude[endTime-startTime,"Seconds"];
  allEnergies  = Flatten[First/@Values[eigensys]];
  groundEnergy = Min[allEnergies];
  eigensys     = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys];
  eigensys     = Association@KeyValueMap[#1->{basis[#1],#2} &,
  eigensys];
  PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
  If[OptionValue["Return as states"],
    (
      PrintFun["> Padding the eigenvectors to correspond to the
  level basis ..."];
      eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values
  [#[[2, 2]] & /@ eigensys]];
      globalBasis   = Flatten[Values[basis], 1];
      eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
      eigenJs       = Flatten[KeyValueMap[ConstantArray[#1, Length
  [#2[[2, 2]]]] &, eigensys]];
      states        = Transpose[{eigenEnergies, eigenJs,
  eigenVectors}];
      states        = SortBy[states, First];
      eigenVectors  = Last /@ states;
      LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
  InputForm[#[[2]]]]) & /@ globalBasis;
      majorComponentIndices = Ordering[Abs[#]][[-1]] & /@
  eigenVectors;
      levelLabels           = LSJmultiplets[[majorComponentIndices
  ]];
      Return[{globalBasis, levelLabels, states}];
    ),
    Return[{basis, eigensys}]
  ];
 )
];
```

2 The coefficients of fractional parentage

In the 1920s and 1930s, when spectroscopic evidence was being studied to inform the nascent 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 $\underline{\ell}^{n-1}$ one can create the wavefunctions for $\underline{\ell}^{n-1}$ with all the required antisymmetry and normalization conditions. In this approach, a given *daughter* term in $\underline{\ell}^{n}$ has a number of *parent* terms in $\underline{\ell}^{n-1}$, with the coefficients of fractional parentage determining how much of each parent is in the daughter as a sum over parents

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

More importantly for **qlanth**, the coefficients of fractional parentage can be used to evaluate matrix elements of operators, such as in &qn-28, &qn-50, &qn-64, 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 $\underline{\ell}^{n}$ configurations, reduced matrix elements in $\underline{\ell}^{3}$ for the three-body operators, and reduced matrix elements in $\underline{\ell}^{2}$ 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 $\underline{\ell}^{n}$ 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 $\underline{f}^7$ (such is the case in `B1f_all.txt`), tabulating these beyond $\underline{f}^7$ would be redundant since the coefficients of fractional

parentage beyond $\underline{\mathrm{f}}^7$ satisfy relationships with those below $\underline{\mathrm{f}}^7$. According to [NK63]

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

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

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

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

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

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

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

3 The JJ' block structure

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

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

Fig-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).

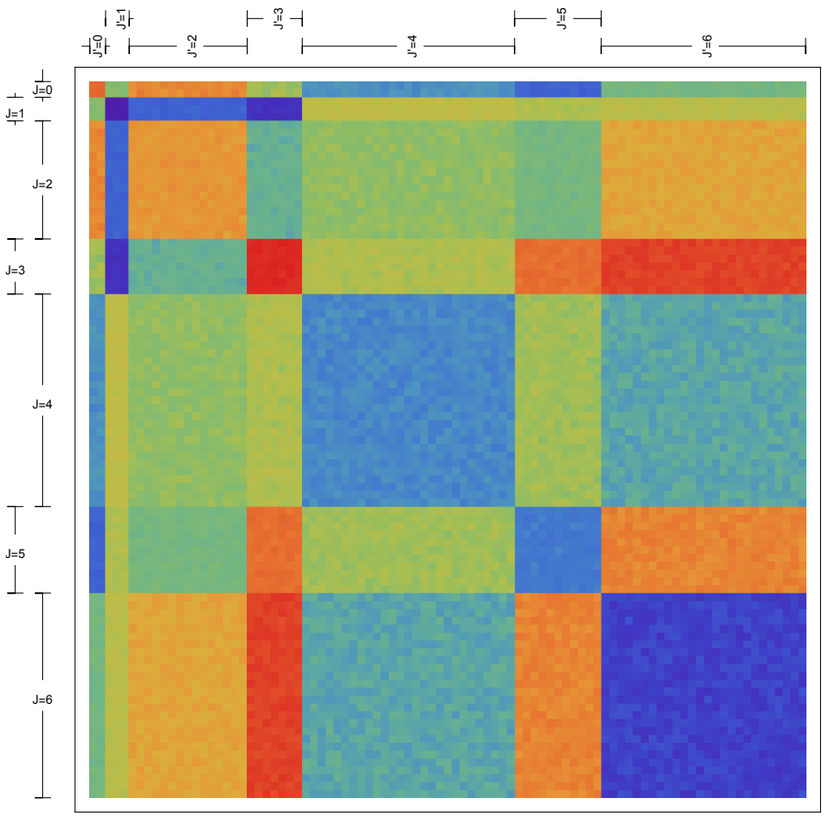


Figure 1: The block structure for $\underline{\mathrm{f}}^2$

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

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

```
Options[JJBlockMatrix] = {"Sparse"->True, "ChenDeltas"->False};
JJBlockMatrix[numE_, J_, Jp_, CFTable_, OptionsPattern[]] := Module[
  {NKSLJMs, NKSLJMps, NKSLJM, NKSLJMp,
  SLterm, SpLpterm,
  MJ, MJp,
  subKron, matValue, eMatrix},
  (
    NKSLJMs  = AllowedNKSLJMforJTerms[numE, J];
    NKSLJMps = AllowedNKSLJMforJTerms[numE, Jp];
    eMatrix =
      Table[
        (*Condition for a scalar matrix op*)
        SLterm   = NKSLJM[[1]];
        SpLpterm = NKSLJMp[[1]];
        MJ       = NKSLJM[[3]];
        MJp      = NKSLJMp[[3]];
        subKron  = (
            KroneckerDelta[J, Jp] *
            KroneckerDelta[MJ, MJp]
                  );
        matValue =
          If[subKron==0,
            0,
              (
                ElectrostaticTable[{numE, SLterm, SpLpterm}] +
                ElectrostaticConfigInteraction[{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 level description of the eigenstates.

```
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 default is False
The option \"Set t2Switch\" can be used to toggle on or off setting
    the t2 selector automatically or not, the default is True, which
    replaces the parameter according to numE.
The option \"ReturnInBlocks\" can be used to return the matrix in
    block or flattened form. The default is to return it in flattened
    form.";
Options[HamMatrixAssembly] = {
      "FilenameAppendix"->"",
      "IncludeZeeman"->False,
      "Set t2Switch"->True,
      "ReturnInBlocks"->False};
HamMatrixAssembly[nf_, OptionsPattern[]] := Module[
  {numE, ii, jj, howManyJs, Js, blockHam},
  (
    (*######################################*)
    ImportFun = ImportMZip;
    (*######################################*)
    (*hole-particle equivalence enforcement*)
```

```
    numE = nf;
    allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2, T2p,
      T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
      α, β, γ, B02, B04, B06, B12, B14, B16,
      B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16, S22,
      S24, S26, S34, S36, S44, S46, S56, S66, T11, T11p, T12, T14, T15, T16,
      T17, T18, T19, Bx, By, Bz};
    params0 = AssociationThread[allVars, allVars];
    If[nf > 7,
      (
        numE = 14 - nf;
        params = HoleElectronConjugation[params0];
        If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
      ),
      params = params0;
      If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
    ];
    (* Load symbolic expressions for LS,J,J' energy sub-matrices. *)
    emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" -> OptionValue["FilenameAppendix"]];
    JJBlockMatrixTable = ImportFun[emFname];
    (*Patch together the entire matrix representation using J,J' blocks.*)
    PrintTemporary["Patching JJ blocks ..."];
    Js         = AllowedJ[numE];
    howManyJs  = Length[Js];
    blockHam   = ConstantArray[0, {howManyJs, howManyJs}];
    Do[
      blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[jj]]}];,
    {ii, 1, howManyJs},
    {jj, 1, howManyJs}
    ];

    (* Once the block form is created flatten it *)
    If[Not[OptionValue["ReturnInBlocks"]],
      (blockHam = ArrayFlatten[blockHam];
      blockHam  = ReplaceInSparseArray[blockHam, params];
      ),
      (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam,{2}];)
    ];

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

4 The effective Hamiltonian

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

To describe the effect of the charges in the lattice surrounding the atom, 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 is of limited applicability if taken too literally, however, if only symmetry considerations are assumed, the model is seen to have greater validity but a somewhat less clear physical origin.

The Hilbert space of a multi-electron ion is a vast stage. In principle 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 57. Furthermore, the exclusion principle also guides us in what type of orbital we could possibly place these three electrons, in the case of the lanthanide ions, this being the 4f orbitals. But not really, there are many more unoccupied orbitals outside of the xenon core, two of these electrons, if they are willing to pay the energetic price, they could find themselves in a 5d or a 6s orbital.

Here we shall assume a single-configuration description. Meaning that all the valence electrons in the ions that we study here will all be considered to be located in f-orbitals, or what is the same, that they are described by $\underline{\mathrm{f}}^n$wavefunctions. This is, however, a harsh approximation, but thankfully one can make some amends to it. The effects that arise in the single configuration description because of omitting all the other possible orbitals where the electrons might find themselves, this is what is called *configuration-interaction*.

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

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

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

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

$$+ \overbrace{\underbrace{\hat{\mathcal{H}}_{\text{e:e}}}_{\text{e:e}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathcal{H}}_{\text{G}_2}}_{\text{G}_2\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(7)}}_{\mathcal{SO}(7)\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\text{Y}}}_{\substack{\text{effective}\\ \text{three-body}}} + \underbrace{\hat{\mathcal{H}}_{\text{cf}}}_{\text{crystal field}}}^{\text{electric interactions}} \tag{6}$$

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

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

$$\hat{\mathcal{H}}_{\text{s:o}} = \begin{cases} \sum_{i=1}^{n}\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \text{ with } \xi(r_i) = \frac{\hbar^2}{2m^2c^2r_i}\frac{\mathrm{d}V_{\text{sn}}(r_i)}{\mathrm{d}r_i} \\ \sum_{i=1}^{n}\zeta\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \text{ with } \zeta \text{ the radial average of } \xi(r_i) \text{ or used as phenomenological parameter} \end{cases} \tag{9}$$

$$\hat{\mathcal{H}}_{\text{s:s}} = \sum_{k=0,2,4} M^{(k)}\hat{m}_k^{ss} \tag{10}$$

$$\hat{\mathcal{H}}_{\text{s:oo}\oplus\text{ecs:o}} = \sum_{k=2,4,6} P^{(k)}\hat{p}_k + \sum_{k=0,2,4} M^{(k)}\hat{m}_k \tag{11}$$

$$\hat{\mathcal{H}}_{\text{Z}} = -\vec{B}\cdot\hat{\mu} = \mu_B\vec{B}\cdot\left(\hat{L} + g_s\hat{S}\right) \text{ (interaction with a magnetic field)} \tag{12}$$

$$\hat{\mathcal{H}}_{\text{e:e}} = \sum_{i>j}^{n,n}\frac{e^2}{\|\vec{r}_i - \vec{r}_j\|} = \sum_{k=0,2,4,6} F^{(k)}\hat{f}_k \text{ (v-electron:v-electron repulsion)} \tag{13}$$

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

$$\hat{\mathcal{H}}_{\mathcal{SO}(3)} = \alpha\,\mathcal{C}(\mathcal{SO}(3)) = \alpha\hat{L}^2 \text{ (Trees effective operator)} \tag{14}$$

$$\hat{\mathcal{H}}_{\text{G}_2} = \beta\,\mathcal{C}(\mathcal{G}_2) \tag{15}$$

$$\hat{\mathcal{H}}_{\mathcal{SO}(7)} = \gamma\,\mathcal{C}(\mathcal{SO}(7)) \tag{16}$$

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

$$\hat{\mathcal{H}}_{\text{cf}} = \sum_{i=1}^{n} V_{\text{CF}}(\hat{r}_i) = \sum_{i=1}^{n}\sum_{k=2,4,6}\sum_{q=-k}^{k}\mathcal{B}_q^{(k)}C_q^{(k)}(i) \text{ (crystal field interaction of v-electrons with electrostatic field due to surroundings)} \tag{18}$$

For concreteness Fig. 2 shows a depiction of a matrix representation of this Hamiltonian for the $\underline{\text{f}}^2$ configuration.

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 having been integrated in the parameters that the effective Hamiltonian has.

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

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

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

4.2 $\hat{\mathcal{H}}_{\text{e:sn}}$: 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

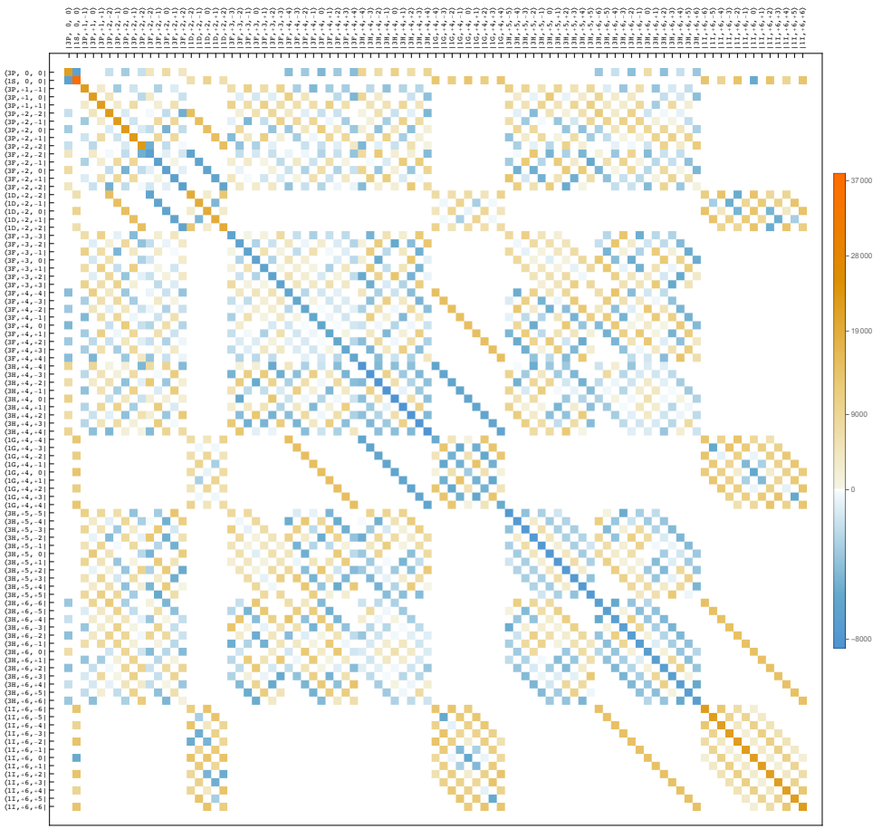


Figure 2: The matrix representation for Pr^{3+} in LaF_3 in the $|LSJM_J\rangle$ basis.

surrounding the nucleus.

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

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

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

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

This term is the first we will not discard. Calculating this term for the $\underline{f}^n$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 $\underline{f}^n$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 $\underline{f}^n$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.

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

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 $\hat{\mathscr{H}}_{\text{s:o}}$: 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

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

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

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

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

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

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]:

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

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 $\underline{\jmath} = \frac{1}{2}$ 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 interac-

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

The calculation of the $\hat{\mathcal{H}}_{\mathrm{s:s\text{-}s:oo}}$ 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 $\underline{\mathrm{f}}^2$ or in $\underline{\mathrm{f}}^3$ and then we "pull" them up for all $\underline{\mathrm{f}}^n$ configurations with help of the coefficients of fractional parentage.

The analysis of *spin-other-orbit*, and the *spin-spin* contributions used in **qlanth** is that of Judd, Crosswhite, and Crosswhite [JCC68]. Much as the spin-orbit effect can be extracted 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

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

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*

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

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

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

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

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

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

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

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

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]

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

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 time as the electrostically-correlated-spin-orbit (see next section).

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

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

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

```
ReducedT22infn::usage = "ReducedT22infn[n, SL, SpLp] calculates the
    reduced matrix element of the T22 operator for the f^n
    configuration corresponding to the terms SL and SpLp. This is 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]
    ]
  )
];
```

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

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

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

where $\xi(r_h)(\underline{\hat{\jmath}}_h\cdot\underline{\hat{\ell}}_h)$ 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, SL, SLp, J}] returns the matrix element of the magnetic interaction between the terms SL and SLp in the f^numE configuration for the given value of J. The interaction is given by the sum of the spin-spin, the spin-other-orbit, and the electrostatically-correlated-spin-orbit interactions.
The part corresponding to the spin-spin interaction is provided by SpinSpin[{numE, SL, SLp, J}].
The part corresponding to SOO and ECSO is provided by the function SOOandECSO[{numE, SL, SLp, J}].
The option \"ChenDeltas\" can be used to include or exclude the Chen deltas from the calculation. The default is to exclude them. If this option is used, then the chenDeltas association needs to be loaded into the session with LoadChen[].";
Options[MagneticInteractions] = {"ChenDeltas" -> False};
MagneticInteractions[{numE_, SL_, SLp_, J_}, OptionsPattern[]] :=
  Module[
  {key, ss, sooandecso, total,
  S, L, Sp, Lp, phase, sixjay,
```

```
  M0v, M2v, M4v,
  P2v, P4v, P6v},
  (
    key         = {numE, SL, SLp, J};
    ss          = \[Sigma]SS * SpinSpinTable[key];
    sooandecso = SOOandECSOTable[key];
    total = ss + sooandecso;
    total = SimplifyFun[total];
    If[
      Not[OptionValue["ChenDeltas"]],
      Return[total]
    ];
    (* In the type A errors the wrong values are different *)
    If[MemberQ[Keys[chenDeltas["A"]], {numE, SL, SLp}],
      (
        {S, L}   = FindSL[SL];
        {Sp, Lp} = FindSL[SLp];
        phase    = Phaser[Sp + L + J];
        sixjay   = SixJay[{Sp, Lp, J}, {L, S, 1}];
        {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SL,
    SLp}]["wrong"];
        total    = (
                       phase * sixjay *
                       (
                         M0v*M0 + M2v*M2 + M4v*M4 +
                         P2v*P2 + P4v*P4 + P6v*P6
                       )
                   );
        total    = wChErrA * total + (1 - wChErrA) * (ss + sooandecso
    )
      )
    ];
    (* In the type B errors the wrong values are zeros all around *)
    If[MemberQ[chenDeltas["B"], {numE, SL, SLp}],
      (
        total  = (1 - wChErrB) * (ss + sooandecso)
      )
    ];
    Return[total];
  )
];
```

```
GenerateSpinOrbitTable::usage = "GenerateSpinOrbitTable[nmax]
    computes the matrix 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 &qn-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 $\hat{\mathcal{H}}_{\text{⅄}}$: 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)

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

where $\hat{t}_2$ and $\hat{t}_{11}$ are operators that have orthogonal alternatives represented by $\hat{t}'_2$ and $\hat{t}'_{11}$ (see

[JS84]). **qlanth** includes the legacy operator $\hat{t}_2$ since it was used for important work during and before the 1980s.

The omission of some indices in this sum has to do with the fact that the way in which these are defined in terms of their index (see [Jud66]) gives rise to two-body operators which can be absorbed by the two-body terms in the Hamiltonian. As such, it is not so much that they are not included, but rather that their effects are considered to be accounted for elsewhere. This is representative of a common feature of configuration interaction: it gives rise to new intra-configuration operators, but it also contributes to already present operators; this makes it harder to approximate the model parameters *ab-initio*, but is not a practical obstacle for the semi-empirical approach (although it certainly complicates the physical interpretation that each parameter has).

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 $\underline{\mathrm{f}}^n$ 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 $\underline{\mathrm{f}}^3$ 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):

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

The sum in this expression runs over the parents in $\underline{\mathrm{f}}^{n-1}$ that are common to both the daughter terms ψ and ψ' in $\underline{\mathrm{f}}^n$. 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 $\underline{\mathrm{f}}^3$ 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 $\underline{\mathrm{f}}^7$ 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 $\underline{\mathrm{f}}^7$ are used. This yields the unexpected result that in the $\underline{\mathrm{f}}^{12}$ 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 simplicity, and to avoid having to explicitly store matrix elements beyond $\underline{\mathrm{f}}^7$ **qlanth** takes the approach of adding a control parameter `t2Switch` which needs to be set to 1 if below or at $\underline{\mathrm{f}}^7$ and set to 0 if above $\underline{\mathrm{f}}^7$.

```
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}"}] = t20p;

  (*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 the $C_q^{(k)}$ are spherical harmonics normalized with the Racah convention

$$C_q^{(k)} = \sqrt{\frac{4\pi}{2k+1}} Y_q^{(k)}. \tag{52}$$

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

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=-k}^{k} \mathcal{A}_q^{(k)} r_i^k C_q^{(k)}(\theta_i, \phi_i). \tag{53}$$

And if we average the radial coordinate,

$$\hat{\mathcal{H}}_{\mathrm{cf}} = \sum_{i=1}^{n} \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{B}_q^{(k)} C_q^{(k)}(i) \tag{54}$$

where the radial average is included as

$$\mathcal{B}_q^{(k)} \coloneqq \mathcal{A}_q^{(k)} \langle 4f | r^k | 4f \rangle. \tag{55}$$

$\mathcal{B}_q^{(k)}$ may be complex in general. However, since the sum in &qn-53 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

$$\mathcal{B}_q^{(k)} = (-1)^q {\mathcal{B}_{-q}^{(k)}}^*. \tag{56}$$

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

$$\hat{\mathcal{H}}_{\mathrm{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=0}^{\infty} \sum_{q=0}^{k} B_q^{(k)} \left(C_q^{(k)} + (-1)^q C_{-q}^{(k)} \right) + iS_q^{(k)} \left(C_q^{(k)} - (-1)^q C_{-q}^{(k)} \right). \tag{57}$$

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):

$$\langle \ell \| \hat{u}^{(k)} \| \ell' \rangle = 1. \tag{58}$$

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

$$\hat{U}^{(k)} = \sum_{i}^{n} \hat{u}_i^{(k)}. \tag{59}$$

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

$$C_q^{(k)} = \langle \underline{\ell} \| C^{(k)} \| \underline{\ell}' \rangle \hat{u}_q^{(k)} = (-1)^{\underline{\ell}} \sqrt{[\![\underline{\ell}]\!][\![\underline{\ell}']\!]} \begin{pmatrix} \underline{\ell} & k & \underline{\ell}' \\ 0 & 0 & 0 \end{pmatrix} \hat{u}_q^{(k)}. \tag{60}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha S L J M_J | \hat{\mathcal{H}}_{\text{cf}} | \underline{\ell}^n \alpha' S L' J' M_{J'} \rangle}^{\text{Wybourne eqn. 6-3}} = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} \mathcal{B}_q^{(k)} \langle \underline{\ell}^n \alpha S L J M_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' S L' J' M_{J'} \rangle \langle \underline{\ell} \| \hat{C}^{(k)} \| \underline{\ell} \rangle \tag{61}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha S L J M_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' S' L' J' M_{J'} \rangle}^{\text{Wybourne eqn. 6-4}} = (-1)^{J - M_J} \begin{pmatrix} J & k & J' \\ -M_J & q & M_{J'} \end{pmatrix} \langle \underline{\ell}^n \alpha S L J \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S' L' \rangle \tag{62}$$

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

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha S L J \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S' L' \rangle}^{\text{Wybourne eqn. 6-5}} = (-1)^{S+L+J'+k} \sqrt{[\![J]\!][\![J']\!]} \times \\ \begin{Bmatrix} J & J' & k \\ L' & L & S \end{Bmatrix} \langle \underline{\ell}^n \alpha S L \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S' L' \rangle. \end{aligned} \tag{63}$$

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 $|\alpha L S\rangle$ and $|\alpha' L' S'\rangle$ from configuration $\underline{\mathrm{f}}^n$:

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha S L \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S' L' \rangle}^{\text{Cowan eqn. 11.53}} = \delta(S, S') n (-1)^{\underline{\ell}+L+k} \sqrt{[\![L]\!][\![L']\!]} \times \\ \sum_{\bar{\alpha} \bar{L} \bar{S}} (-1)^{\bar{L}} \begin{Bmatrix} \underline{\ell} & k & \underline{\ell} \\ L & \bar{L} & L' \end{Bmatrix} \left(\underline{\ell}^n \alpha L S \left\{ \underline{\ell}^{n-1} \bar{\alpha} \bar{L} \bar{S} \right) \left(\underline{\ell}^{n-1} \bar{\alpha} \bar{L} \bar{S} \right\} \underline{\ell}^n \alpha' L' S' \right). \end{aligned} \tag{64}$$

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 $\triangle(3, k, 3)$ the non-zero contributions only come from $k = 0, 1, 2, 3, 4, 5, 6$. An additional selection rule on k comes from considerations of parity. Since both the bra and the ket in $\langle \underline{\ell}^n \alpha S L J M_J | \hat{\mathcal{H}}_{\text{cf}} | \underline{\ell}^n \alpha' S L' J' M_{J'} \rangle$ have the same parity, then the overall parity of the braket is determined by the parity of $C_q^{(k)}$, 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:

$$\hat{\mathcal{H}}_{\text{cf}}(\vec{r}) = \sum_{i=1}^{n} \sum_{k=2,4,6} \sum_{q=0}^{k} B_q^{(k)} \left(C_q^{(k)} + (-1)^q C_{-q}^{(k)} \right) + i S_q^{(k)} \left(C_q^{(k)} - (-1)^q C_{-q}^{(k)} \right). \tag{65}$$

The above equations are implemented in **qlanth** by the function `CrystalField`. This function puts together the symbolic sum in &qn-61 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];
]
```

Each of the 32 crystallographic point groups requires only a limited number of non-zero crystal field parameters. In **qlanth** these can be queried with the use of the function `CrystalFieldForm`. These were taken from a table in Benelli and Gatteschi [BG15].

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

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

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

This uses data from table 3.3 in Benelli and Gatteschi, 2015.";
CrystalFieldForm[symmetryGroupString_] := (
  If[Not@ValueQ[crystalFieldFunctionalForms],
    crystalFieldFunctionalForms = Import[FileNameJoin[{moduleDir, "
    data", "crystalFieldFunctionalForms.m"}]];
  ];
  crystalFieldFunctionalForms[symmetryGroupString]
)
```

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

$$\hat{\mu} = -\mu_{\mathrm{B}} \left(\hat{L} + g_s \hat{S}\right)^{(1)}, \text{with } \mu_{\mathrm{B}} = 1/2. \tag{66}$$

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]

$$\langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle = \delta(\alpha LSJ, \alpha' L'S'J') \sqrt{J(J+1)(2J+1)} + \delta(\alpha LS, \alpha' L'S') (-1)^{L+S+J+1} \sqrt{[\![J]\!][\![J]\!]} \begin{Bmatrix} L & S & J \\ 1 & J' & S \end{Bmatrix} \tag{67}$$

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

through Wigner-Eckart

$$\langle \alpha LSJM_J | \left(\hat{L} + g_s \hat{S}\right)^{(1)}_q | \alpha' L'S'J'M_{J'} \rangle = (-1)^{J-M_J} \begin{pmatrix} J & 1 & J' \\ -M_J & q & M'_J \end{pmatrix} \langle \alpha LSJ \| \left(\hat{L} + g_s \hat{S}\right)^{(1)} \| \alpha' L'S'J' \rangle \quad (68)$$

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

```
JJBlockMagDip::usage = "JJBlockMagDip[numE, J, Jp] returns an array
    for the LSJM matrix elements of the magnetic dipole operator
    between states with given J and Jp. The option \"Sparse\" can be
    used to return a sparse matrix. The default is to return a sparse
    matrix.
See eqn 15.7 in TASS.
Here it is provided in atomic units in which the Bohr magneton is
    1/2.
\[Mu] = -(1/2) (L + gs S)
We are using the Racah convention for the reduced matrix elements in
    the Wigner-Eckart theorem. See TASS eqn 11.15.
";
Options[JJBlockMagDip]={"Sparse"->True};
JJBlockMagDip[numE_, braJ_, ketJ_, OptionsPattern[]] := Module[
  {braSLJs, ketSLJs,
  braSLJ,   ketSLJ,
  braSL,    ketSL,
  braS,     braL,
  ketS,     ketL,
  braMJ,    ketMJ,
  matValue, magMatrix,
  summand1, summand2,
  threejays},
  (
    braSLJs  = AllowedNKSLJMforJTerms[numE,braJ];
    ketSLJs  = AllowedNKSLJMforJTerms[numE,ketJ];
    magMatrix  = Table[
      braSL     = braSLJ[[1]];
      ketSL     = ketSLJ[[1]];
      {braS, braL}   = FindSL[braSL];
      {ketS, ketL}   = FindSL[ketSL];
      braMJ       = braSLJ[[3]];
      ketMJ       = ketSLJ[[3]];
      summand1    = If[Or[braJ  != ketJ,
                          braSL != ketSL],
        0,
        Sqrt[braJ*(braJ+1)*TPO[braJ]]
      ];
      (* looking at the string includes checking L=L', S=S', and \
    alpha=\alpha'*)
      summand2 = If[braSL!= ketSL,
        0,
        (gs-1) *
          Phaser[braS+braL+ketJ+1] *
            Sqrt[TPO[braJ]*TPO[ketJ]] *
              SixJay[{braJ,1,ketJ},{braS,braL,braS}] *
                Sqrt[braS(braS+1)TPO[braS]]
      ];
      matValue = summand1 + summand2;
      (* We are using the Racah convention for red matrix elements in
     Wigner-Eckart *)
      threejays = (ThreeJay[{braJ, -braMJ}, {1, #}, {ketJ, ketMJ}] &)
     /@ {-1,0,1};
      threejays *= Phaser[braJ-braMJ];
      matValue  = - 1/2 * threejays * matValue;
      matValue,
    {braSLJ, braSLJs},
    {ketSLJ, ketSLJs}
    ];
    If[OptionValue["Sparse"],
      magMatrix = SparseArray[magMatrix]
    ];
    Return[magMatrix];
  )
];
```

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

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

$$\hat{\mu}_y = i\frac{\hat{\mu}^{(1)}_{-1} + \hat{\mu}^{(1)}_{+1}}{\sqrt{2}} \tag{70}$$

$$\hat{\mu}_z = \hat{\mu}^{(1)}_0 \tag{71}$$

```
MagDipoleMatrixAssembly::usage = "MagDipoleMatrixAssembly[numE]
    returns the matrix representation of the operator - 1/2 (L + gs S)
     in the f^numE configuration. The function returns a list with
    three elements corresponding to the x,y,z components of this
    operator. The option \"FilenameAppendix\" can be used to append a
    string to the filename from which the function imports from in
    order to patch together the array. For numE beyond 7 the function
    returns the same as for the complementary configuration. The
    option \"ReturnInBlocks\" can be used to return the matrices in
    blocks. The default is to return the matrices in flattened form
    and as sparse array.";
Options[MagDipoleMatrixAssembly]={
  "FilenameAppendix"->"",
  "ReturnInBlocks"->False};
MagDipoleMatrixAssembly[nf_Integer, OptionsPattern[]] := Module[
  {ImportFun, numE, appendTo,
  emFname, JJBlockMagDipTable,
  Js, howManyJs, blockOp,
  rowIdx, colIdx},
  (
    ImportFun = ImportMZip;
    numE      = nf;
    numH      = 14 - numE;
    numE      = Min[numE,numH];

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

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

Using the cartesian components of the magnetic dipole operator, the matrix elements of the Zeeman term can then be evaluated. This term can be included in the Hamiltonian through an option in 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 energy pseudo-unit Kayser (cm^{-1}). The conversion factor is called `teslaToKayser` in the file `qonstants.m`.

4.11 Going beyond $\underline{f}^7$

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.

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):

$$\zeta, T^{(2)}, T^{(3)}, T^{(4)}, T^{(6)}, T^{(7)}, T^{(8)}, B_q^{(k)} \tag{72}$$

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

```
HoleElectronConjugation::usage = "HoleElectronConjugation[params]
    takes the parameters (as an association) that define a
    configuration and converts them so that they may be interpreted as
     corresponding to a complentary hole configuration. Some of this
    can be simply done by changing the sign of the model parameters.
    In the case of the effective three body interaction the
    relationship is more complex and is controlled by the value of the
     isE variable.";
HoleElectronConjugation[params_] := Module[
  {newparams = params},
  (
    flipSignsOf = {ζ, T2, T3, T4, T6, T7, T8};
    flipSignsOf = Join[flipSignsOf, cfSymbols];
    flipped =
      Table[(flipper -> - newparams[flipper]),
      {flipper, flipSignsOf}
      ];
    nonflipped =
      Table[(flipper -> newparams[flipper]),
      {flipper, Complement[Keys[newparams], flipSignsOf]}
      ];
    flippedParams = Association[Join[nonflipped, flipped]];
    flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
    Return[flippedParams];
  )
];
```

5 Transitions

5.1 State description

5.1.1 Magnetic dipole transitions

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

$$\hat{S}(\psi, \psi') \coloneqq \left|\langle\psi|\hat{\mu}|\psi'\rangle\right|^2 = \left|\langle\psi|\hat{\mu}_x|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_y|\psi'\rangle\right|^2 + \left|\langle\psi|\hat{\mu}_z|\psi'\rangle\right|^2 \tag{73}$$

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

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

```
Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI",
    "States" -> All};
MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern[]]
     := Module[
  {allEigenvecs, Sx, Sy, Sz, Stot, factor},
  (
    numE = Min[14-numE0, numE0];
    (*If not loaded then load it, *)
    If[Or[
        Not[MemberQ[Keys[magOp], numE]],
        OptionValue["Reload MagOp"]],
    (
      magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
    MagDipoleMatrixAssembly[numE];
    )
    ];
    allEigenvecs = Transpose[Last /@ theEigensys];
    Which[OptionValue["States"] === All,
      (
        {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
    allEigenvecs) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      ),
      IntegerQ[OptionValue["States"]],
      (
        singleState  = theEigensys[[OptionValue["States"],2]];
        {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
    singleState) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      )
    ];
    Which[
      OptionValue["Units"] == "SI",
        Return[4 \[Mu]B^2 * Stot],
      OptionValue["Units"] == "Hartree",
        Return[Stot],
      True,
      (
        Print["Invalid option for \"Units\". Options are \"SI\" and \
    "Hartree\"."];
        Abort[];
      )
    ];
  )
];
```

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

$$\mathrm{A_{MD}}\left(|\psi_i\rangle \rightsquigarrow |\psi_f\rangle\right) = \frac{16\pi^3\mu_0}{3h}\frac{n^3}{\lambda_{if}^3}\frac{\hat{S}(\psi_i,\psi_f)}{g_i}, \tag{74}$$

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

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

```
MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]] :=
  Module[
 {AMD, Stot, eigenEnergies,
 transitionWaveLengthsInMeters, nRefractive},
 (
  nRefractive    = OptionValue["RefractiveIndex"];
  numE           = Min[14-numE0, numE0];
  Stot           = MagDipLineStrength[eigenSys, numE, "Units" ->
  OptionValue["Units"]];
  eigenEnergies = Chop[First/@eigenSys];
  energyDiffs    = Outer[Subtract, eigenEnergies, eigenEnergies];
  energyDiffs    = ReplaceDiagonal[energyDiffs, Indeterminate];
  (* Energies assumed in kayser.*)
  transitionWaveLengthsInMeters = 0.01/energyDiffs;

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

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

$$\mathrm{f}_{\mathrm{MD}}\left(|\psi_g\rangle \rightsquigarrow |\psi_e\rangle\right) = \frac{8\pi^2 m_e}{3\,hc\,e^2}\frac{n}{\lambda_{ge}}\frac{\hat{S}(\psi_g,\psi_e)}{g_g} \tag{75}$$

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

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

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

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

5.2 Level description

5.2.1 Forced electric dipole transitions

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

However, much as configuration interaction is an essential component in the description of the electronic structure, it having a bearing on the energy spectrum and the intra-configuration wavefunctions themselves. Configuration interaction may also be used to bring back into the analysis the fact that the *actual* wavefunctions will also have at least a small part of them in other configurations, even if most of them may be within the ground configuration. It is therefore the case that the *actual* parity of the wavefunctions is mixed, and therefore intra-configuration [2] electric dipole transitions are actually allowed. These electric dipole transitions are called *forced* electric dipole transitions.

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

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

$$\mathrm{f}_{\text{f-ED}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S'L'J'\rangle\right) = \mathcal{R}\frac{8\pi^2 m_e}{3h}\frac{\nu}{2J+1}\frac{\chi}{n}\sum_{k=2,4,6}\Omega_{(k)}\left|\langle \underline{\mathrm{f}}^n \alpha SLJ\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha' S'L'J'\rangle\right|^2, \tag{76}$$

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

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

$$\chi = \frac{n^2+2}{3} \tag{77}$$

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

5.2.2 Magnetic dipole transitions

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

$$\hat{\mathcal{S}}(|\alpha LSJ\rangle, |\alpha' S'L'J'\rangle) = \left|\langle \alpha LSJ\|\frac{1}{2}(\hat{L} + g\hat{S})\|\alpha' S'L'J'\rangle\right|^2 \tag{78}$$

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

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

```
LevelMagDipoleLineStrength::usage = "LevelMagDipoleLineStrength[
    eigenSys, numE] calculates the magnetic dipole line strengths for
    an ion whose level description is determined by levelParams. The
    function returns a square array whose elements represent the
    magnetic dipole line strengths between the levels given in
    eigenSys such that the element magDipoleLineStrength[[i,j]] is the
     line strength between the levels |Subscript[\[Psi], i]> and |
    Subscript[\[Psi], j]>. Eigensys must be such that it consists of a
     lists of lists where in each list the last element corresponds to
     the eigenvector of a level (given as a row) in the standard basis
     for levels of the f^numE configuration.
The function admits the following options:
  \"Units\" : The units in which the line strengths are given. The
    default is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI
    \" then the unit of the line strength is (A m^2)^2 = (J/T)^2. If \
    "Hartree\" then the line strength is given in units of 2 \[Mu]B.";
Options[LevelMagDipoleLineStrength] = {
  "Units"->"SI"
  };
LevelMagDipoleLineStrength[theEigensys_List, numE0_Integer,
    OptionsPattern[]] := Module[
  {numE, levelMagOp, allEigenvecs, magDipoleLineStrength, units},
  (
    numE          = Min[14-numE0,numE0];
    levelMagOp    = LevelMagDipoleMatrixAssembly[numE];
    allEigenvecs  = Transpose[Last/@theEigensys];
    units         = OptionValue["Units"];
    magDipoleLineStrength          = Transpose[allEigenvecs].
    levelMagOp.allEigenvecs;
    magDipoleLineStrength          = Abs[magDipoleLineStrength]^2;
    Which[
      units=="SI",
        Return[4 \[Mu]B^2 * magDipoleLineStrength],
      units=="Hartree",
        Return[magDipoleLineStrength]
    ];
  )
];
```

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

$$\mathrm{f_{MD}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S' L' J'\rangle\right) = \frac{2n}{3}\frac{\mathcal{E}(|\alpha' S' L' J'\rangle) - \mathcal{E}(|\alpha LSJ)\rangle}{2J+1}\alpha^2 \hat{\mathcal{S}}\left(|\alpha LSJ\rangle, |\alpha' S' L' J'\rangle\right) \tag{79}$$

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

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

```
LevelMagDipoleOscillatorStrength::usage = "
    LevelMagDipoleOscillatorStrength[eigenSys, numE] calculates the
    magnetic dipole oscillator strengths for an ion whose level
    description is determined by levelParams. The refractive index of
    the medium is relevant, but here it is assumed to be 1, this can
    be changed through the option \"RefractiveIndex\". eigenSys must
    consist of a lists of lists with three elements: the first element
     being the energy of the level, the second element being the J of
    the level, and the third element being the eigenvector of the
    level.
  The function returns a list with the following elements:
    - basis : A list with the allowed {SL, J} terms in the f^numE
    configuration. Equal to BasisLSJ[numE].
    - eigenSys : A list with the eigensystem of the Hamiltonian for
    the f^n configuration in the level description.
    - levelLabels : A list with the labels of the major components of
     the calculated levels.
    - magDipoleOstrength : A square array whose elements represent
    the magnetic dipole oscillator strengths between the levels given
    in eigenSys such that the element magDipoleOstrength[[i,j]] is the
     oscillator strength between the levels |Subscript[\[Psi], i]> and
     |Subscript[\[Psi], j]>. In this array the elements below the
    diagonal represent emission oscillator strengths, and elements
    above the diagonal represent absorption oscillator strengths. The
```

```
    emission oscillator strengths are negative. The oscillator
    strength is a dimensionless quantity.
  The function admits the following option:
    \"RefractiveIndex\" : The refractive index of the medium where
    the transitions are taking place. This may be a number or a
    function. If a number then the oscillator strengths are calculated
     assuming a wavelength-independent refractive index as given. If a
     function then the refractive indices are calculated accordingly
    to the vaccum wavelength of each transition (the function must
    admit a single argument equal to the wavelength in nm). The
    default is 1.
  For reference see equation (27.8) in Rudzikas (2007). The
    expression for the line strenght is the simplest when using atomic
     units, (27.8) is missing a factor of α^2.";
Options[LevelMagDipoleOscillatorStrength]={
  "RefractiveIndex" -> 1
};
LevelMagDipoleOscillatorStrength[eigenSys_, numE_, OptionsPattern[]]
    := Module[
  {eigenEnergies, eigenVecs, levelJs,
  energyDiffs, magDipoleOstrength, nRef,
  wavelengthsInNM, nRefs, degenDivisor},
  (
    basis          = BasisLSJ[numE];
    eigenEnergies  = First/@eigenSys;
    nRef           = OptionValue["RefractiveIndex"];
    eigenVecs      = Last/@eigenSys;
    levelJs        = #[[2]]&/@eigenSys;
    energyDiffs    = -Outer[Subtract,eigenEnergies,eigenEnergies];
    energyDiffs   *= kayserToHartree;
    magDipoleOstrength = LevelMagDipoleLineStrength[eigenSys, numE, "
    Units"->"Hartree"];
    magDipoleOstrength = 2/3 * αFine^2 * energyDiffs *
    magDipoleOstrength;
    degenDivisor       = #1 / ( 2 * levelJs[[#2[[1]]]] + 1 ) &;
    magDipoleOstrength = MapIndexed[degenDivisor, magDipoleOstrength,
     {2}];
    Which[nRef===1,
      True,
      NumberQ[nRef],
        (
          magDipoleOstrength = nRef * magDipoleOstrength;
        ),
      True,
        (
          wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
    10^7;
          nRefs              = Map[nRef, wavelengthsInNM];
          magDipoleOstrength = nRefs * magDipoleOstrength;
        )
    ];
    Return[{basis, eigenSys, magDipoleOstrength}];
  )
];
```

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

$$\Gamma_{\mathrm{MD}}\left(|\alpha LSJ\rangle \rightsquigarrow |\alpha' S' L' J'\rangle\right) = \frac{4n^3}{3}\frac{\left(\mathcal{E}(|\alpha LSJ\rangle) - \mathcal{E}(|\alpha' S' L' J'\rangle)\right)^3}{2J+1}\alpha^5 \hat{\mathcal{S}}\left(|\alpha LSJ\rangle, |\alpha' S' L' J'\rangle\right). \tag{80}$$

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

```
LevelMagDipoleSpotaneousDecayRates::usage = "
    LevelMagDipoleSpotaneousDecayRates[eigenSys, numE] calculates the
    spontaneous emission rates for the magnetic dipole transitions
    between the levels given in eigenSys. The function returns a
    square array whose elements represent the spontaneous emission
    rates between the levels given in eigenSys such that the element
    [[i,j]] of the returned array is the rate of spontaneous emission
    from the level |Subscript[\[Psi], i]> to the level |Subscript[\[
    Psi], j]>. In this array the elements below the diagonal represent
     emission rates, and elements above the diagonal are identically
    zero.
The function admits two optional arguments:
  + \"Units\" : The units in which the rates are given. The default
    is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI\" then
```

```
    the rates are given in s^-1. If \"Hartree\" then the rates are
    given in the atomic unit of frequency.
  + \"RefractiveIndex\" : The refractive index of the medium where
    the transitions are taking place. This may be a number or a
    function. If a number then the rates are calculated assuming a
    wavelength-independent refractive index as given. If a function
    then the refractive indices are calculated accordingly to the
    vaccum wavelength of each transition (the function must admit a
    single argument equal to the wavelength in nm). The default is 1."
    ;
Options[LevelMagDipoleSpotaneousDecayRates] = {
  "Units" -> "SI",
  "RefractiveIndex" -> 1};
LevelMagDipoleSpotaneousDecayRates[eigenSys_List, numE_Integer,
    OptionsPattern[]] := Module[
  {
    levMDlineStrength, eigenEnergies, energyDiffs, levelJs,
    spontaneousRatesInHartree, spontaneousRatesInSI, degenDivisor,
    units,
    nRef, nRefs, wavelengthsInNM
  },
  (
    nRef               = OptionValue["RefractiveIndex"];
    units              = OptionValue["Units"];
    levMDlineStrength = LowerTriangularize@LevelMagDipoleLineStrength
    [eigenSys,numE,"Units"->"Hartree"];
    levMDlineStrength = SparseArray[levMDlineStrength];
    eigenEnergies      = First /@ eigenSys;
    energyDiffs        = Outer[Subtract, eigenEnergies, eigenEnergies
    ];
    energyDiffs        = kayserToHartree * energyDiffs;
    energyDiffs        = SparseArray[LowerTriangularize[energyDiffs]];
    levelJs            = #[[2]]& /@ eigenSys;
    spontaneousRatesInHartree = 4/3 αFine^5 * energyDiffs^3 *
    levMDlineStrength;
    degenDivisor               = #1 / (2*levelJs[[#2[[1]]]] + 1)&;
    spontaneousRatesInHartree = MapIndexed[degenDivisor,
    spontaneousRatesInHartree, {2}];
    Which[nRef===1,
      True,
      NumberQ[nRef],
        (
          spontaneousRatesInHartree = nRef^3 *
    spontaneousRatesInHartree;
        ),
      True,
        (
          wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
    10^7;
          nRefs              = Map[nRef, wavelengthsInNM];
          spontaneousRatesInHartree = nRefs^3 *
    spontaneousRatesInHartree;
        )
    ];
    If[units == "SI",
      (
        spontaneousRatesInSI = 1/hartreeTime *
    spontaneousRatesInHartree;
        Return[SparseArray@spontaneousRatesInSI];
      ),
      Return[SparseArray@spontaneousRatesInHartree];
    ];
  )
];
```

6 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. **Level Calculation**: Determines the level description using free-ion parameters.

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

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

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

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

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

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

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

- `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 level calculation.
- truncationEnergy: the energy level at which the Hamiltonian was truncated.
- numE: the number of electrons in the f^{numE} configuration.
- expData: the experimental data used for the fitting process.
- problemVars: the variables considered during the fitting process.
- maxIterations: the maximum number of iterations used in the fitting process.
- hamDim: the dimension of the full Hamiltonian before simplifications or truncations.
- allVars: all the symbols defining the Hamiltonian under the applied simplifications.
- freeBies: the free-ion parameters used to calculate the intermediate coupling levels.
- truncatedDim: the dimension of the truncated Hamiltonian.

- `compiledIntermediateFname`: the file name of the compiled function used for the truncated Hamiltonian.
- `fittedLevels`: the number of levels that were fitted.
- `actualSteps`: the actual number of steps taken by the fitting algorithm.
- `solWithUncertainty`: a list of replacement rules showing the best fit value and its uncertainty for each parameter.
- `rmsHistory`: Aa list of the RMS values found during the fitting process.
- `Appendix`: an association appended to the log file under the "Appendix" key.
- `presentDataIndices`: the indices in `expData` that were used for fitting.
- `states`: a list of eigenvalues and eigenvectors for the fitted model, available if eigenvectors were saved.
- `energies`: a list of the energies of the fitted levels, adjusted if an energy shift was included in the fitting.

Table Fig-3 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 3: 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.

```

```
7 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\".
\"presentDataIndices\" which is the list of indices in expData that
    were used for fitting, this takes into account both the empty
    indices in expData and also the indices in excludeDataIndices.

\"states\" which contains a list of eigenvalues and eigenvectors for
    the fitted model, this is only available if the option \"
    SaveEigenvectors\" is set to True; if a general shift of energy
    was allowed for in the fitting, then the energies are shifted
    accordingly.
\"energies\" which is a list of the energies of the fitted levels,
    this is only available if the option \"SaveEigenvectors\" is set
    to False. If a general shift of energy was allowed for in the
    fitting, then the energies are shifted accordingly.

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

  \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
    5.
  \"TrucationEnergy\": if Automatic then the maximum energy in
    expData is taken, else it takes the value set by this option. In
    all cases the energies in expData are only considered up to this
    value.
  \"PrintFun\": the function used to print progress messages, the
    default is PrintTemporary.

  \"SlackChannel\": name of the Slack channel to which to dump
    progress messaages, the default is None which disables this option
    .
  \"ProgressView\": whether or not a progress window will be opened
    to show the progress of the solver, the default is True.
  \"SignatureCheck\": if True then then the function returns
    prematurely, returning a list with the symbols that would have
    defined the Hamiltonian after all simplifications have been
    applied. Useful to check the entire parameter set that the
    Hamiltonian has, which has to match one-to-one what is provided by
     startingValues.
  \"SaveEigenvectors\": if True then the both the eigenvectors and
    eigenvalues are saved under the \"states\" key of the returned
    association. If False then only the energies are saved, the
    default is False.

  \"AppendToLogFile\": an association appended to the log file under
    the key \"Appendix\".
  \"MagneticSimplifier\": a list of replacement rules to simplify the
     Marvin and pesudo-magnetic paramters. Here the ratios of the
    Marvin parameters and the pseudo-magnetic parameters are defined
    to simplify the magnetic part of the Hamiltonian.
  \"MagFieldSimplifier\": a list of replacement rules to specify a
    magnetic field (in T), if set to {}, then {Bx, By, Bz} can also be
     used as variables to be fitted for.

  \"SymmetrySimplifier\": a list of replacements rules to simplify
    the crystal field.
  \"OtherSimplifier\": an additional list of replacement rules that
    are applied to the Hamiltonian before computing with it. Here the
    spin-spin contribution can be turned off by setting \[Sigma]SS->0,
     which is the default.
";
Options[ClassicalFit] = {
  "MaxHistory"       -> 200,
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "calcs",
  "ProgressView"     -> True,
```

```
  "TruncationEnergy" -> Automatic,
  "AccuracyGoal"     -> 5,
  "PrintFun"         -> PrintTemporary,
  "SlackChannel"     -> None,
  "ProgressView"     -> True,
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "MagneticSimplifier" -> {
    M2 -> 56/100 M0,
    M4 -> 31/100 M0,
    P4 -> 1/2 P2,
    P6 -> 1/10 P2
    },
  "MagFieldSimplifier" -> {
    Bx -> 0,
    By -> 0,
    Bz -> 0
    },
  "SymmetrySimplifier" -> {
    B12->0, B14->0, B16->0, B34->0, B36->0, B56->0,
    S12->0 ,S14->0, S16->0, S22->0, S24->0, S26->0,
    S34->0 ,S36->0, S44->0, S46->0, S56->0, S66->0
    },
  "OtherSimplifier" -> {
    F0->0,
    P0->0,
    \[Sigma]SS->0,
    T11p->0, T11->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0
    },
  "ThreeBodySimplifier" -> <|
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0},
    2 -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      },
    3  -> {},
    4  -> {},
    5  -> {},
    6  -> {},
    7  -> {},
    8  -> {},
    9  -> {},
    10 -> {},
    11 -> {},
    12 -> {
      T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      },
    13->{
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      }
    |>,
    "FreeIonSymbols" -> {F0, F2, F4, F6, ζ}
};
ClassicalFit[numE_Integer, expData_List, excludeDataIndices_List,
    problemVars_List, startValues_Association, \[Sigma]exp_?NumericQ,
    constraints_List, OptionsPattern[]]:=Module[
  {accuracyGoal, activeVarIndices, activeVars, activeVarsString,
    activeVarsWithRange, allFreeEnergies, allFreeEnergiesSorted,
    allVars, allVarsVec, argsForEvalInsideOfTheIntermediateSystems,
    argsOfTheIntermediateEigensystems, aVar, aVarPosition, basis,
    basisChanger, basisChangerBlocks, bestError, bestParams, bestRMS,
```

```
    blockShifts, blockSizes, colIdx, compiledDiagonal,
    compiledIntermediateFname, constrainedProblemVars,
    constrainedProblemVarsList, covMat, currentRMS, degressOfFreedom,
    dependentVars, diagonalBlocks, diagonalScalarBlocks, diff,
    eigenEnergies, eigenvalueDispenserTemplate, eigenVectors,
    elevatedIntermediateEigensystems, endTime, fmSol, fmSolAssoc,
    fractionalWidth, freeBies, freeIenergiesAndMultiplets,
    freeionSymbols, fullHam, fullSolVec, funcString, ham, hamDim,
    hamEigenvaluesTemplate, hamString, hess, indepSolVecVec, indepVars
    , intermediateHam, isolationValues, jobVars, lin, linMat, ln,
    lnParams, logFilePrefix, logFname, magneticSimplifier,
    maxFreeEnergy, maxHistory, maxIterations, methodString,
    methodStringTemplate, minFreeEnergy, minpoly, modelSymbols,
    multipletAssignments, needlePosition, numBlocks, numQSignature,
    numReps, solCompendium, openNotebooks, ordering, othersFixed,
    otherSimplifier, p0, paramBest, paramSigma, perHam, polySols,
    presentDataIndices, PrintFun, problemVarsPositions, problemVarsQ,
    problemVarsQString, problemVarsVec, problemVarsWithStartValues,
    reducedModelSymbols, resultMessage, roundedTruncationEnergy,
    rowIdx, runningInteractive, shiftToggle, simplifier, slackChan,
    sol, solAssoc, sols, solWithUncertainty, sortedTruncationIndex,
    sqdiff, standardValues, starTime, startingValues, startTime,
    startVarValues, states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, thisPoly, threadHeaderTemplate
    , threadMessage, threadTS, timeTaken, totalVariance,
    truncadedFname, truncatedIntermediateBasis,
    truncatedIntermediateHam, truncationEnergy, truncationIndices,
    truncationUmbral, usingInitialRange, varHash, varIdx,
    varsWithConstants, varWithValsSignature, \[Lambda]0Vec, \[Lambda]
    exp},
143 (
144   solCompendium   = <||>;
145   addShift      = OptionValue["AddConstantShift"];
146   ln            = theLanthanides[[numE]];
147   maxHistory    = OptionValue["MaxHistory"];
148   maxIterations = OptionValue["MaxIterations"];
149   logFilePrefix = If[OptionValue["FilePrefix"] == "",
150                      ToString[theLanthanides[[numE]]],
151                      OptionValue["FilePrefix"]
152                    ];
153   accuracyGoal = OptionValue["AccuracyGoal"];
154   slackChan    = OptionValue["SlackChannel"];
155   PrintFun     = OptionValue["PrintFun"];
156   freeIonSymbols      = OptionValue["FreeIonSymbols"];
157   runningInteractive  = (Head[$ParentLink] === LinkObject);
158   magneticSimplifier  = OptionValue["MagneticSimplifier"];
159   magFieldSimplifier  = OptionValue["MagFieldSimplifier"];
160   symmetrySimplifier  = OptionValue["SymmetrySimplifier"];
161   otherSimplifier     = OptionValue["OtherSimplifier"];
162   threeBodySimplifier = If[Head[OptionValue["ThreeBodySimplifier"]]
      == Association,
163                            OptionValue["ThreeBodySimplifier"][numE],
164                            OptionValue["ThreeBodySimplifier"]
165                          ];
166
167   truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic
      ,
168     PrintFun["Truncation energy set to Automatic, using the maximum
       energy in the data ..."];
169     Max[Select[First /@ expData, NumericQ[#] &]],
170     OptionValue["TruncationEnergy"]
171     ];
172   PrintFun["Using a truncation energy of ", truncationEnergy, " K"
     ];
173
174   simplifier          = Join[magneticSimplifier,
175                              magFieldSimplifier,
176                              symmetrySimplifier,
177                              threeBodySimplifier,
178                              otherSimplifier];
179
180   PrintFun["Determining gaps in the data ..."];
181   (* the indices that are numeric in expData whatever is non-
      numeric is assumed as a known gap *)
182   presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
```

```
    , ___}]];
183 (* some indices omitted here based on the excludeDataIndices
    argument *)
184 presentDataIndices = Complement[presentDataIndices,
    excludeDataIndices];
185
186 hamDim                              = Binomial[14, numE];
187 solCompendium["simplifier"]          = simplifier;
188 solCompendium["excludeDataIndices"] = excludeDataIndices;
189 solCompendium["startValues"]         = startValues;
190 solCompendium["freeIonSymbols"]      = freeIonSymbols;
191 solCompendium["truncationEnergy"]    = truncationEnergy;
192 solCompendium["numE"]                = numE;
193 solCompendium["expData"]             = expData;
194 solCompendium["problemVars"]         = problemVars;
195 solCompendium["maxIterations"]       = maxIterations;
196 solCompendium["hamDim"]              = hamDim;
197 solCompendium["constraints"]         = constraints;
198 modelSymbols  = Sort[Select[paramSymbols, Not[MemberQ[Join[
    racahSymbols, juddOfeltIntensitySymbols, chenSymbols,{t2Switch, \[
    Epsilon],gs}],#]]&]];
199 (* remove the symbols that will be removed by the simplifier, no
    symbol should remain here that is not in the symbolic Hamiltonian
    *)
200 reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
    simplifier],#]]&];
201
202 (* this is useful to understand what are the arguments of the
    truncated compiled Hamiltonian *)
203 If[OptionValue["SignatureCheck"],
204   (
205     Print["Given the model parameters and the simplifying
    assumptions, the resultant model parameters are:"];
206     Print[{reducedModelSymbols}];
207     Print["Exiting ..."];
208     Return[""];
209   )
210 ];
211
212 (* calculate the basis *)
213 PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
214 basis    = BasisLSJMJ[numE];
215
216 (* get the reference parameters from LaF3 *)
217 PrintFun["Getting reference free-ion parameters for ",ln," using
    LaF3 ..."];
218 lnParams = LoadParameters[ln];
219 freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
      numE];
220 (* a more explicit alias *)
221 allVars           = reducedModelSymbols;
222 numericConstraints = Association@Select[constraints, NumericQ
    [#[[2]]] &];
223 standardValues    = allVars /. Join[lnParams, numericConstraints
    ];
224 solCompendium["allVars"]  = allVars;
225 solCompendium["freeBies"] = freeBies;
226
227 (* reload compiled version if found *)
228 varHash                  = Hash[{numE, allVars, freeBies,
    truncationEnergy, simplifier}];
229 compiledIntermediateFname = ln<>"-compiled-intermediate-truncated
    -ham-"<>ToString[varHash]<>".mx";
230 compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
    compiledIntermediateFname}];
231 solCompendium["compiledIntermediateFname"] =
    compiledIntermediateFname;
232
233 If[FileExistsQ[compiledIntermediateFname],
234   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 ..."];
235   PrintFun["Using : ", compiledIntermediateFname];
236   {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
     Import[compiledIntermediateFname];,
```

```
237     (
238       (* grab the Hamiltonian preserving its block structure *)
239       PrintFun["Assembling the Hamiltonian for f^",numE," keeping the
      block structure ..."];
240       ham          = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
241       (* apply the simplifier *)
242       PrintFun["Simplifying using the aggregate set of simplification
      rules ..."];
243       ham          = Map[ReplaceInSparseArray[#, simplifier]&, ham,
    {2}];
244       PrintFun["Zeroing out every symbol in the Hamiltonian that is
    not a free-ion parameter ..."];
245       (* Get the free ion symbols *)
246       freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
      freeIonSymbols];
247       (* Take the diagonal blocks for the intermediate analysis *)
248       PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."
    ];
249       diagonalBlocks       = Diagonal[ham];
250       (* simplify them to only keep the free ion symbols *)
251       PrintFun["Simplifying the diagonal blocks to only keep the free
      ion symbols ..."];
252       diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
    ]&/@diagonalBlocks;
253       (* these include the MJ quantum numbers, remove that *)
254       PrintFun["Contracting the basis vectors by removing the MJ
    quantum numbers from the diagonal blocks ..."];
255       diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
256
257       argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
    Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
258       argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
    ToString[#]<>"v") & /@ freeIonSymbols,", "]];
259       PrintFun["argsOfTheIntermediateEigensystems = ",
    argsOfTheIntermediateEigensystems];
260       PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
    argsForEvalInsideOfTheIntermediateSystems];
261       PrintFun["(if the following fails, it might help to see if the
    arguments of TheIntermediateEigensystems match the ones shown
    above)"];
262
263       (* 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 *)
264       (* compile one function for each of the blocks *)
265       PrintFun["Compiling functions for the diagonal blocks of the
    Hamiltonian ..."];
266       compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N
    [Normal[#]]]]&/@diagonalScalarBlocks;
267       (* use that to create a function that will calculate the free-
    ion eigensystem *)
268       TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζv_
    ] := (
269         theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&)/
    @compiledDiagonal;
270         theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
271         Js      = AllowedJ[numEv];
272         basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
273         (* having calculated the eigensystems with the removed
    degeneracies, put the degeneracies back in explicitly *)
274         elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2
    Js[[#2[[1]]]]+1]&, theIntermediateEigensystems];
275         (* Identify a single MJ to keep *)
276         pivot = If[EvenQ[numEv],0,-1/2];
277         LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
    @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
278         (* calculate the multiplet assignments that the intermediate
    basis eigenvectors have *)
279         needlePosition = 0;
280         multipletAssignments = Table[
281           (
282             J          = Js[[idx]];
283             eigenVecs = theIntermediateEigensystems[[idx]][[2]];
284             majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
    @eigenVecs;
285             majorComponentIndices       += needlePosition;
```

```
286             needlePosition               += Length[
    majorComponentIndices];
287             majorComponentAssignments    = LSJmultiplets[[#]]&/
    @majorComponentIndices;
288             (* All of the degenerate eigenvectors belong to the same
    multiplet*)
289             elevatedMultipletAssignments = ListRepeater[
    majorComponentAssignments,2J+1];
290             elevatedMultipletAssignments
291           ),
292         {idx, 1, Length[Js]}
293         ];
294         (* put together the multiplet assignments and the energies *)
295         freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
296         freeIenergiesAndMultiplets = Flatten[
    freeIenergiesAndMultiplets, 1];
297         (* calculate the change of basis matrix using the
    intermediate coupling eigenvectors *)
298         basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
299         basisChanger = SparseArray[basisChanger];
300         Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
301       );
302
303       PrintFun["Calculating the intermediate eigensystems for ",ln,"
    using free-ion params from LaF3 ..."];
304       (* calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
305       {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
306
307       (* use that intermediate coupling basis to compile a function
    for the full Hamiltonian *)
308       allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
309       (* important that the intermediate coupling basis have attached
     energies, which make possible the truncation *)
310       ordering = Ordering[allFreeEnergies];
311       (* sort the free ion energies and determine which indices
    should be included in the truncation *)
312       allFreeEnergiesSorted          = Sort[allFreeEnergies];
313       {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
314       (* determine the index at which the energy is equal or larger
    than the truncation energy *)
315       sortedTruncationIndex = Which[
316         truncationEnergy > (maxFreeEnergy-minFreeEnergy),
317         hamDim,
318         True,
319         FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted
    ],x_/;x>truncationEnergy,{0},1][[1]]
320       ];
321       (* the actual energy at which the truncation is made *)
322       roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
323
324       (* the indices that enact the truncation *)
325       truncationIndices = ordering[[;;sortedTruncationIndex]];
326       (* Return[{basisChanger, ham, truncationIndices}]; *)
327       PrintFun["Computing the block structure of the change of basis
    array ..."];
328       blockSizes = BlockArrayDimensionsArray[ham];
329       basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
330       blockShifts = First /@ Diagonal[blockSizes];
331       numBlocks = Length[blockSizes];
332       (* using the ham (with all the symbols) change the basis to the
     computed one *)
333       PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
334       (* intermediateHam            = Transpose[basisChanger].ham.
    basisChanger; *)
335       (* Return[{basisChangerBlocks, ham}]; *)
336       intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks];
```

```
337       PrintFun["Distributing products inside of symbolic matrix
      elements to keep complexity in check ..."];
338       Do[
339         intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
      intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
340         {rowIdx, 1, numBlocks},
341         {colIdx, 1, numBlocks}
342       ];
343       intermediateHam = BlockMatrixMultiply[BlockTranspose[
      basisChangerBlocks],intermediateHam];
344       PrintFun["Distributing products inside of symbolic matrix
      elements to keep complexity in check ..."];
345       Do[
346         intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
      intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
347         {rowIdx, 1, numBlocks},
348         {colIdx, 1, numBlocks}
349       ];
350       (* using the truncation indices truncate that one *)
351       PrintFun["Truncating the Hamiltonian ..."];
352       truncatedIntermediateHam = TruncateBlockArray[intermediateHam,
      truncationIndices, blockShifts];
353       (* these are the basis vectors for the truncated hamiltonian *)
354       PrintFun["Saving the truncated intermediate basis ..."];
355       truncatedIntermediateBasis = basisChanger[[All,
      truncationIndices]];
356
357       PrintFun["Compiling a function for the truncated Hamiltonian
      ..."];
358       (* compile a function that will calculate the truncated
      Hamiltonian given the parameters in allVars, this is the function
      to be use in fitting *)
359       compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
      Evaluate[truncatedIntermediateHam]];
360       (* save the compiled function *)
361       PrintFun["Saving the compiled function for the truncated
      Hamiltonian and the truncated intermediate basis ..."];
362       Export[compiledIntermediateFname, {
      compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
363     )
364     ];
365
366     truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
367     PrintFun["The truncated Hamiltonian has a dimension of ",
      truncationUmbral, "x", truncationUmbral, " ..."];
368     presentDataIndices = Select[presentDataIndices, # <=
      truncationUmbral &];
369     solCompendium["presentDataIndices"] = presentDataIndices;
370
371     (* the problemVars are the symbols that will be fitted for *)
372
373     PrintFun["Starting up the fitting process using the Levenberg-
      Marquardt method ..."];
374     (* using the problemVars I need to create the argument list
      including _?NumericQ *)
375     problemVarsQ = (ToString[#] <> "_?NumericQ") & /@ problemVars;
376     problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
377     (* we also need to have the padded arguments with the variables
      in the right position and the fixed values in the remaining ones
      *)
378     problemVarsPositions = Position[allVars, #][[1, 1]] & /@
      problemVars;
379     problemVarsString = StringJoin[Riffle[ToString /@ problemVars, ",
       "]];
380     (* 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 *)
381     varsWithConstants = standardValues;
382     varsWithConstants[[problemVarsPositions]] = problemVars;
383     varsWithConstantsString = ToString[varsWithConstants];
384
385     (* this following function serves eigenvalues from the
      Hamiltonian, has memoization so it might grow to use a lot of RAM
      *)
386     Clear[HamSortedEigenvalues];
387     hamEigenvaluesTemplate = StringTemplate["
```

```
388     HamSortedEigenvalues[`problemVarsQ`]:=(
389       ham          = compileIntermediateTruncatedHam@@`
        varsWithConstants`;
390       eigenValues = Sort@Eigenvalues@ham;
391       eigenValues = eigenValues - Min[eigenValues];
392       HamSortedEigenvalues[`problemVarsString`] = eigenValues;
393       Return[eigenValues]
394     )"];
395     hamString = hamEigenvaluesTemplate[<|
396         "problemVarsQ" -> problemVarsQString,
397         "varsWithConstants" -> varsWithConstantsString,
398         "problemVarsString" -> problemVarsString
399         |>];
400     ToExpression[hamString];
401
402     (* 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 *)
403     eigenvalueDispenserTemplate = StringTemplate["
404     PartialHamEigenvalues[`problemVarsQ`][i_]:=(
405       eigenVals = HamSortedEigenvalues[`problemVarsString`];
406       eigenVals[[i]]
407     )
408     "];
409     eigenValueDispenserString =
410       eigenvalueDispenserTemplate[<|
411         "problemVarsQ"      -> problemVarsQString,
412         "problemVarsString" -> problemVarsString
413         |>];
414     ToExpression[eigenValueDispenserString];
415
416     PringFun["Determining the free variables after constraints ..."];
417     constrainedProblemVars = (problemVars /. constraints);
418     constrainedProblemVarsList = Variables[constrainedProblemVars];
419     If[addShift,
420       PrintFun["Adding a constant shift to the fitting parameters ...
        "];
421       constrainedProblemVarsList = Append[constrainedProblemVarsList,
        \[Epsilon]]
422     ];
423
424     indepVars = Complement[problemVars, #[[1]] & /@ constraints];
425     stringPartialVars = ToString/@constrainedProblemVarsList;
426
427     paramSols  = {};
428     rmsHistory = {};
429     steps      = 0;
430     problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
431     If[addShift,
432       problemVarsWithStartValues = Append[problemVarsWithStartValues,
        {\[Epsilon] ,0}];
433     ];
434     openNotebooks     = If[runningInteractive,
435               ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
        [],
436               {}];
437     If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
        ProgressView"],
438       ProgressNotebook["Basic"->False]
439     ];
440     degressOfFreedom = Length[presentDataIndices] - Length[
        problemVars] - 1;
441     PrintFun["Fitting for ", Length[presentDataIndices], " data
        points with ", Length[problemVars], " free parameters.", " The
        effective degrees of freedom are ", degressOfFreedom, " ..."];
442
443     PrintFun["Starting the fitting process ..."];
444     startTime=Now;
445     shiftToggle = If[addShift, 1, 0];
446     sol = FindMinimum[
447       Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@
        constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
448       {j, presentDataIndices}],
449       problemVarsWithStartValues,
450       Method        -> "LevenbergMarquardt",
451       MaxIterations -> OptionValue["MaxIterations"],
```

```
452       AccuracyGoal  -> OptionValue["AccuracyGoal"],
453       StepMonitor :> (
454         steps      += 1;
455         currentRMS = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues
    @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j,
    presentDataIndices}];
456         currentRMS = Sqrt[currentRMS / degressOfFreedom];
457         paramSols  = AddToList[paramSols, constrainedProblemVarsList,
     maxHistory];
458         rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
459         )
460       ];
461     endTime = Now;
462     timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
463     PrintFun["Solution found in ", timeTaken, "s"];
464
465     solVec = constrainedProblemVars /. sol[[-1]];
466     indepSolVec = indepVars /. sol[[-1]];
467     If[addShift,
468       \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
469       \[Epsilon]Best = 0
470     ];
471     fullSolVec = standardValues;
472     fullSolVec[[problemVarsPositions]] = solVec;
473     PrintFun["Calculating the numerical Hamiltonian corresponding to
    the solution ..."];
474     fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
475     PrintFun["Calculating energies and eigenvectors ..."];
476     {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
477     states = Transpose[{eigenEnergies, eigenVectors}];
478     states = SortBy[states, First];
479     eigenEnergies = First /@ states;
480     PrintFun["Shifting energies to make ground state zero of energy
    ..."];
481     eigenEnergies = eigenEnergies - eigenEnergies[[1]];
482     PrintFun["Calculating the linear approximant to each eigenvalue
    ..."];
483     allVarsVec = Transpose[{allVars}];
484     p0 = Transpose[{fullSolVec}];
485     linMat = {};
486     If[addShift,
487       tail = -2,
488       tail = -1];
489     Do[
490       (
491         aVarPosition = Position[allVars, aVar][[1, 1]];
492         isolationValues = ConstantArray[0, Length[allVars]];
493         isolationValues[[aVarPosition]] = 1;
494         dependentVars = KeyValueMap[{#1, D[#2, aVar]} &, Association[
    constraints]];
495         Do[
496           isolationValues[[Position[allVars, dVar[[1]]][[1, 1]]]] =
    dVar[[2]],
497           {dVar, dependentVars}
498         ];
499         perHam  = compileIntermediateTruncatedHam @@ isolationValues;
500         lin     = FirstOrderPerturbation[Last /@ states, perHam];
501         linMat  = Append[linMat, lin];
502       ),
503       {aVar, constrainedProblemVarsList[[;;tail]]}
504     ];
505     PrintFun["Removing the gradient of the ground state ..."];
506     linMat = (# - #[[1]] & /@ linMat);
507     PrintFun["Transposing derivative matrices into columns ..."];
508     linMat  = Transpose[linMat];
509
510     PrintFun["Calculating the eigenvalue vector at solution ..."];
511     \[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
512     PrintFun["Putting together the experimental vector ..."];
513     \[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
    ]]}];
514     problemVarsVec = If[addShift,
515       Transpose[{constrainedProblemVarsList[[;;-2]]}],
516       Transpose[{constrainedProblemVarsList}]
517     ];
518     indepSolVecVec = Transpose[{indepSolVec}];
```

```
    PrintFun["Calculating the difference between eigenvalues at solution ..."];
    diff = If[linMat=={},
      (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
      (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[presentDataIndices]].(problemVarsVec - indepSolVecVec)
    ];
    PrintFun["Calculating the sum of squares of differences around solution ... "];
    sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
    PrintFun["Calculating the minimum (which should coincide with sol ..."];
    minpoly       = sqdiff /. AssociationThread[problemVars -> solVec];
    fmSolAssoc    = Association[sol[[2]]];
    totalVariance = Length[presentDataIndices]*\[Sigma]exp^2;
    PrintFun["Calculating the uncertainty in the parameters ..."];
    solWithUncertainty = Table[
      (
        aVar        = constrainedProblemVarsList[[varIdx]];
        paramBest   = aVar /. fmSolAssoc;
        othersFixed = AssociationThread[Delete[constrainedProblemVarsList[[;;tail]], varIdx] -> Delete[indepSolVec, varIdx]];
        thisPoly    = sqdiff /. othersFixed;
        polySols    = Last /@ Last /@ Solve[thisPoly == minpoly + 1*totalVariance];
        polySols    = Select[polySols, Im[#] == 0 &];
        paramSigma  = Max[polySols] - Min[polySols];
        (aVar -> {paramBest, paramSigma})
      ),
    {varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
    ];
    PrintFun["Calculating the covariance matrix ..."];
    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];
  )
];
```

7 Accompanying notebooks

qlanth is accompanied by the following auxiliary Mathematica notebooks:

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

8 Additional data

8.1 Carnall et al data on Ln:LaF_3

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

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

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

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

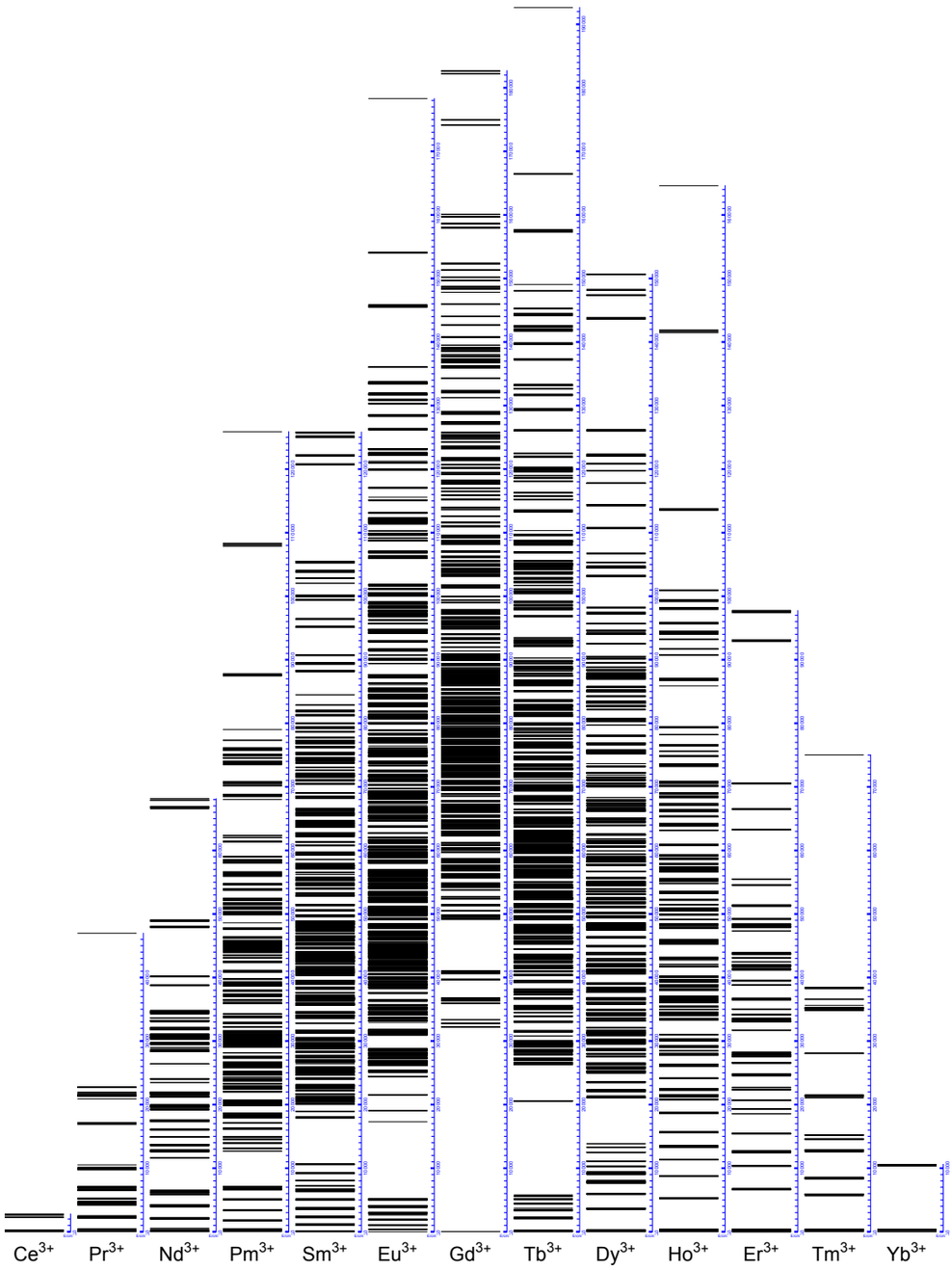


Figure 4: Dieke plot for lanthanum fluoride from data generated by **qlanth**.

```
    ];
    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-72 and setting t2Switch to 0. For configurations at or below $\underline{f}^7$ it is necessary to set t2Switch to 1.

9 Units

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

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

```
BeginPackage["qonstants`"];

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

(* Spectroscopic niceties*)
theLanthanides = {"Ce", "Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho", "Er", "Tm", "Yb"};
theActinides   = {"Ac", "Th", "Pa", "U", "Np", "Pu", "Am", "Cm", "Bk", "Cf", "Es", "Fm", "Md", "No", "Lr"};
theTrivalents  = {"Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho", "Er", "Tm"};
specAlphabet   = "SPDFGHIKLMNOQRTUV"

(* SI *)
hPlanck = 6.62607015 * 10^-34;          (* Planck's constant in J s *)
hBar    = hPlanck    / (2 \[Pi]);       (* reduced Planck's constant in J s *)
\[Mu]B  = 9.2740100783 * 10^-24;        (* Bohr magneton in SI *)
me      = 9.1093837015 * 10^-31;        (* electron mass in kg *)
cLight  = 2.99792458   * 10^8;          (* speed of light in m/s *)
eCharge = 1.602176634  * 10^-19;        (* elementary charge in SI *)
\[Epsilon]0  = 8.8541878128 * 10^-12;   (* electric permittivity in vacuum in SI *)
\[Mu]0  = 4 \[Pi] * 10^-7;              (* magnetic permeability in vacuum in SI *)
αFine = 1/137.036;                (* fine structure constant *)

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

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

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

EndPackage[];
```

10 Notation

$$\overbrace{\hat{l}}^{\text{orbital angular momentum operator of a single electron}} \tag{81}$$

$$\overbrace{\hat{L}}^{\text{total orbital angular momentum operator}} \tag{82}$$

$$\overbrace{\hat{s}}^{\text{spin angular momentum operator of a single electron}} \tag{83}$$

$$\overbrace{\hat{S}}^{\text{total spin angular momentum operator}} \tag{84}$$

$$\overbrace{\Lambda}^{\text{Shorthand for all other quantum numbers}} \tag{85}$$

$$\overbrace{\underline{\ell}}^{\text{orbital angular momentum number}} \tag{86}$$

$$\overbrace{\underline{s}}^{\text{spinning angular momentum number}} \tag{87}$$

$$\overbrace{\hat{\mathfrak{C}}}^{\text{Coulomb non-central potential}} \tag{88}$$

$$\overbrace{\langle \Lambda LS \| \hat{O} \| \Lambda' L' S' \rangle}^{\text{LS-reduced matrix element of operator } \hat{O} \text{ between } \Lambda LS \text{ and } \Lambda' L' S'} \tag{89}$$

$$\overbrace{\langle \Lambda LSJ \| \hat{O} \| \Lambda' L' S' J' \rangle}^{\text{LSJ-reduced matrix element of operator } \hat{O} \text{ between } \Lambda LSJ \text{ and } \Lambda' L' S' J'} \tag{90}$$

$$\overbrace{{}^{2S+1}\alpha \mathrm{L} \equiv |\alpha LS\rangle}^{\text{Spectroscopic term } \alpha LS \text{ in Russel-Saunders notation}} \tag{91}$$

$$\overbrace{\hat{X}^{(k)}}^{\text{spherical tensor operator of rank k}} \tag{92}$$

$$\overbrace{\hat{X}^{(k)}_q}^{\text{q-component of the spherical tensor operator } \hat{X}^{(k)}} \tag{93}$$

$$\overbrace{\left(\underline{\ell}^{n-1} \alpha' L' S' \middle\} \underline{\ell}^n \alpha LS\right)}^{\text{The coefficient of fractional parentage from the parent term } |\underline{\ell}^{n-1} \alpha' L' S'\rangle \text{ for the daughter term } |\underline{\ell}^n \alpha LS\rangle} \tag{94}$$

11 Definitions

$$\overbrace{[\![x]\!] \coloneqq 2x+1}^{\text{two plus one}} \tag{95}$$

$$\overbrace{\hat{u}^{(k)}}^{\text{irreducible unit tensor operator of rank k}} \tag{96}$$

$$\overbrace{\hat{U}^{(k)} \coloneqq \sum_{i=1}^{n} \hat{u}^{(k)}}^{\text{symmetric unit tensor operator for n equivalent electrons}} \tag{97}$$

$$\overbrace{\mathrm{C}^{(k)}_q \coloneqq \sqrt{\frac{4\pi}{2k+1}} \mathrm{Y}^{(k)}_q}^{\text{Renormalized spherical harmonics}} \tag{98}$$

$$\overbrace{\triangleleft(j_1, j_2, j_3) \coloneqq \begin{cases} 1 & \text{if } j_1 = (j_2 + j_3), (j_2 + j_3 - 1), \ldots, |j_2 - j_3| \\ 0 & \text{otherwise} \end{cases}}^{\text{Triangle “delta” between } j_1, j_2, j_3} \tag{99}$$

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 and expanded 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 many of the
functions included in this module with some simple use cases.

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

REFERENCES:

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

+ Rudzikas, Zenonas. Theoretical Atomic Spectroscopy, 2007.

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

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

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

ζ: spin-orbit strength parameter.

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

```
144
145 M0: 0th Marvin integral
146 M2: 2nd Marvin integral
147 M4: 4th Marvin integral
148 \[Sigma]SS: spin-spin override, if 0 spin-spin is omitted, if 1 then
    spin-spin is included
149
150 T2:  three-body effective operator parameter T^2 (non-orthogonal)
151 T2p: three-body effective operator parameter T^2' (orthogonalized T2)
152 T3:  three-body effective operator parameter T^3
153 T4:  three-body effective operator parameter T^4
154 T6:  three-body effective operator parameter T^6
155 T7:  three-body effective operator parameter T^7
156 T8:  three-body effective operator parameter T^8
157
158 T11:  three-body effective operator parameter T^11
159 T11p: three-body effective operator parameter T^11'
160 T12:  three-body effective operator parameter T^12
161 T14:  three-body effective operator parameter T^14
162 T15:  three-body effective operator parameter T^15
163 T16:  three-body effective operator parameter T^16
164 T17:  three-body effective operator parameter T^17
165 T18:  three-body effective operator parameter T^18
166 T19:  three-body effective operator parameter T^19
167
168 P0: pseudo-magnetic parameter P^0
169 P2: pseudo-magnetic parameter P^2
170 P4: pseudo-magnetic parameter P^4
171 P6: pseudo-magnetic parameter P^6
172
173 gs: electronic gyromagnetic ratio
174
175 α: Trees' parameter α describing configuration interaction via the
    Casimir operator of SO(3)
176 β: Trees' parameter β describing configuration interaction via the
    Casimir operator of G(2)
177 γ: Trees' parameter γ describing configuration interaction via the
    Casimir operator of SO(7)
178
179 B02: crystal field parameter B_0^2 (real)
180 B04: crystal field parameter B_0^4 (real)
181 B06: crystal field parameter B_0^6 (real)
182 B12: crystal field parameter B_1^2 (real)
183 B14: crystal field parameter B_1^4 (real)
184
185 B16: crystal field parameter B_1^6 (real)
186 B22: crystal field parameter B_2^2 (real)
187 B24: crystal field parameter B_2^4 (real)
188 B26: crystal field parameter B_2^6 (real)
189 B34: crystal field parameter B_3^4 (real)
190
191 B36: crystal field parameter B_3^6 (real)
192 B44: crystal field parameter B_4^4 (real)
193 B46: crystal field parameter B_4^6 (real)
194 B56: crystal field parameter B_5^6 (real)
195 B66: crystal field parameter B_6^6 (real)
196
197 S12: crystal field parameter S_1^2 (real)
198 S14: crystal field parameter S_1^4 (real)
199 S16: crystal field parameter S_1^6 (real)
200 S22: crystal field parameter S_2^2 (real)
201
202 S24: crystal field parameter S_2^4 (real)
203 S26: crystal field parameter S_2^6 (real)
204 S34: crystal field parameter S_3^4 (real)
205 S36: crystal field parameter S_3^6 (real)
206
207 S44: crystal field parameter S_4^4 (real)
208 S46: crystal field parameter S_4^6 (real)
209 S56: crystal field parameter S_5^6 (real)
210 S66: crystal field parameter S_6^6 (real)
211
212 \[Epsilon]: ground level baseline shift
213 t2Switch: controls the usage of the t2 operator beyond f7 (1 for f7
    or below, 0 for f8 or above)
214 wChErrA: If 1 then the type-A errors in Chen are used, if 0 then not.
```

```
wChErrB: If 1 then the type-B errors in Chen are used, if 0 then not.

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

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

nE: number of electrons in a configuration
";
paramSymbols    = StringSplit[paramAtlas, "\n"];
paramSymbols    = Select[paramSymbols, # != ""& ];
paramSymbols    = ToExpression[StringSplit[#, ":"][[1]]] & /@
    paramSymbols;
Protect /@ paramSymbols;

(* Parameter families *)
Unprotect[racahSymbols, chenSymbols, slaterSymbols, controlSymbols,
    cfSymbols, TSymbols, pseudoMagneticSymbols, marvinSymbols,
    casimirSymbols, magneticSymbols, juddOfeltIntensitySymbols];
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};
juddOfeltIntensitySymbols = {\[CapitalOmega]2, \[CapitalOmega]4, \[
    CapitalOmega]6};
paramFamilies         = Hold[{racahSymbols, chenSymbols,
    slaterSymbols, controlSymbols, cfSymbols, TSymbols,
    pseudoMagneticSymbols, marvinSymbols, casimirSymbols,
    magneticSymbols, juddOfeltIntensitySymbols}];
ReleaseHold[Protect /@ paramFamilies];
crystalGroups = {"C1","Ci","C2","Cs","C2h","D2","C2v","D2h","C4","S4"
    ,"C4h","D4","C4v","D3d","D4h","C3","C3i","D3","C3v","D3d","C6","
    C3h","C6h","D6","C6v","D3h","D6h","T","Th","O","Td","Oh"};

(* Parameter usage *)
paramLines = Select[StringSplit[paramAtlas, "\n"], # != "" &];
usageTemplate = StringTemplate["`paramSymbol`::usage=\"`paramSymbol`
    : `paramUsage`\";"];
Do[(
  {paramString, paramUsage} = StringSplit[paramLine, ":"];
  paramUsage                = StringTrim[paramUsage];
  expressionString          = usageTemplate[<|"paramSymbol" ->
    paramString, "paramUsage" -> paramUsage|>];
  ToExpression[usageTemplate[<|"paramSymbol" -> paramString, "
    paramUsage" -> paramUsage|>]]
),
{paramLine, paramLines}
];

AllowedJ;
AllowedMforJ;
AllowedNKSLJMforJMTerms;
AllowedNKSLJMforJTerms;
AllowedNKSLJTerms;

AllowedNKSLTerms;
AllowedNKSLforJTerms;
AllowedSLJMTerms;
AllowedSLJTerms;
AllowedSLTerms;

BasisLSJ;
```

```
BasisLSJMJ;
BasisTableGenerator;
Bqk;
CFP;

CFPAssoc;
CFPTable;
CFPTerms;
Carnall;
CasimirG2;

CasimirSO3;
CasimirSO7;
Cqk;
CrystalField;
CrystalFieldForm;

Dk;
EigenLever;
Electrostatic;
ElectrostaticConfigInteraction;
ElectrostaticTable;

EnergyLevelDiagram;
EnergyStates;
EtoF;
ExportMZip;
ExportmZip;

FindNKLSTerm;
FindSL;
FreeHam;
FromArrayToTable;
FtoE;

GG2U;
GSO7W;
GenerateCFP;
GenerateCFPAssoc;
GenerateCFPTable;

GenerateCrystalFieldTable;
GenerateElectrostaticTable;
GenerateReducedUkTable;
GenerateReducedV1kTable;
GenerateSOOandECSOLSTable;

GenerateSOOandECSOTable;
GenerateSpinOrbitTable;
GenerateSpinSpinTable;
GenerateT22Table;
GenerateThreeBodyTables;

GenerateThreeBodyTables;
Generator;
GroundMagDipoleOscillatorStrength;
HamMatrixAssembly;
HamiltonianForm;

HamiltonianMatrixPlot;
HoleElectronConjugation;
ImportMZip;
IonSolver;
JJBlockMagDip;

JJBlockMatrix;
JJBlockMatrixFileName;
JJBlockMatrixTable;
JuddOfeltUkSquared;
LabeledGrid;

LevelElecDipoleOscillatorStrength;
LevelJJBlockMagDipole;
LevelMagDipoleLineStrength;
LevelMagDipoleMatrixAssembly;
LevelMagDipoleOscillatorStrength;
```

```
352
353 LevelMagDipoleSpontaneousDecayRates;
354 LevelSimplerSymbolicHamMatrix;
355 LevelSolver;
356 ListRepeater;
357 LoadAll;
358
359 LoadCFP;
360 LoadCarnall;
361 LoadChenDeltas;
362 LoadElectrostatic;
363 LoadGuillotParameters;
364
365 LoadParameters;
366 LoadSOOandECSO;
367 LoadSOOandECSOLS;
368 LoadSpinOrbit;
369 LoadSpinSpin;
370
371 LoadSymbolicHamiltonians;
372 LoadT11;
373 LoadT22;
374 LoadTermLabels;
375 LoadThreeBody;
376
377 LoadUk;
378 LoadV1k;
379 MagDipLineStrength;
380 MagDipoleMatrixAssembly;
381 MagDipoleRates;
382
383 MagneticInteractions;
384 MapToSparseArray;
385 MaxJ;
386 MinJ;
387 NKCFPPhase;
388
389 ParamPad;
390 ParseBenelli2015;
391 ParseStates;
392 ParseStatesByNumBasisVecs;
393 ParseStatesByProbabilitySum;
394
395 ParseTermLabels;
396 Phaser;
397 PrettySaunders;
398 PrettySaundersSLJ;
399 PrettySaundersSLJmJ;
400
401 PrintL;
402 PrintSLJ;
403 PrintSLJM;
404 ReducedSOOandECSOinf2;
405 ReducedSOOandECSOinfn;
406
407 ReducedT11inf2;
408 ReducedT22inf2;
409 ReducedT22infn;
410 ReducedUk;
411 ReducedUkTable;
412
413 ReducedV1kTable;
414 Reducedt11inf2;
415 ReplaceInSparseArray;
416 SOOandECSO;
417 SOOandECSOLSTable;
418
419 SOOandECSOTable;
420 ScalarOperatorProduct;
421 Seniority;
422 ShiftedLevels;
423 SimplerSymbolicHamMatrix;
424
425 SixJay;
426 SpinOrbit;
427 SpinOrbitTable;
```

```
SpinSpin;
SpinSpinTable;

Sqk;
SquarePrimeToNormal;
TPO;
TabulateJJBlockMagDipTable;
TabulateJJBlockMatrixTable;

TabulateManyJJBlockMagDipTables;
TabulateManyJJBlockMatrixTables;
ThreeBodyTable;
ThreeBodyTables;
ThreeJay;

TotalCFIters;
chenDeltas;
fK;
fnTermLabels;
fsubk;

fsupk;
moduleDir;
symbolicHamiltonians;

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

Begin["`Private`"]

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

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

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

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

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

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

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

```
489
490 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.
491 The function additionally takes an option \"Print\" that if set to
      True, will print the symbols that were not present in the given
      association. The default is True.";
492 Options[ParamPad] = {"Print" -> True};
493 ParamPad[params_, OptionsPattern[]] := (
494   notPresentSymbols = Complement[paramSymbols, Keys[params]];
495   If[OptionValue["Print"],
496     Print["Following symbols were not given and are being set to 0:
      ",
497     notPresentSymbols]
498   ];
499   newParams = Transpose[{paramSymbols, ConstantArray[0, Length[
      paramSymbols]]}];
500   newParams = (#[[1]] -> #[[2]]) & /@ newParams;
501   newParams = Association[newParams];
502   newParams = Join[newParams, params];
503   Return[newParams];
504   )
505
506 (* ############################################################ *)
507 (* ###################### Racah Algebra ###################### *)
508
509 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.";
510 ReducedUk[numE_, l_, SL_, SpLp_, k_] := Module[
511   {spin, orbital, Uk, S, L,
512   Sp, Lp, Sb, Lb, parentSL,
513   cfpSL, cfpSpLp, Ukval,
514   SLparents, SLpparents,
515   commonParents, phase},
516     {spin, orbital} = {1/2, 3};
517     {S, L}          = FindSL[SL];
518     {Sp, Lp}        = FindSL[SpLp];
519     If[Not[S == Sp],
520       Return[0]
521     ];
522     cfpSL      = CFP[{numE, SL}];
523     cfpSpLp    = CFP[{numE, SpLp}];
524     SLparents  = First /@ Rest[cfpSL];
525     SLpparents = First /@ Rest[cfpSpLp];
526     commonParents = Intersection[SLparents, SLpparents];
527     Uk = Sum[(
528       {Sb, Lb} = FindSL[\[Psi]b];
529       Phaser[Lb] *
530         CFPAssoc[{numE, SL, \[Psi]b}] *
531         CFPAssoc[{numE, SpLp, \[Psi]b}] *
532         SixJay[{orbital, k, orbital}, {L, Lb, Lp}]
533     ),
534     {\[Psi]b, commonParents}
535     ];
536     phase     = Phaser[orbital + L + k];
537     prefactor = numE * phase * Sqrt[TPO[L,Lp]];
538     Ukval     = prefactor*Uk;
539     Return[Ukval];
540 ]
541
542 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'.";
543 Ck[orbital_, k_] := (-1)^orbital * TPO[orbital] * ThreeJay[{orbital
      , 0}, {k, 0}, {orbital, 0}];
544
545 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.";
546 SixJay[{j1_, j2_, j3_}, {j4_, j5_, j6_}] := (
547   sixJayval = Which[
```

```
      Not[TriangleAndSumCondition[j1, j2, j3]],
      0,
      Not[TriangleAndSumCondition[j1, j5, j6]],
      0,
      Not[TriangleAndSumCondition[j4, j2, j6]],
      0,
      Not[TriangleAndSumCondition[j4, j5, j3]],
      0,
      True,
      SixJSymbol[{j1, j2, j3}, {j4, j5, j6}]];
   SixJay[{j1, j2, j3}, {j4, j5, j6}] = sixJayval);

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

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

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.";
Options[GenerateReducedUkTable] = {"Export" -> True, "Progress" ->
  True};
GenerateReducedUkTable[numEmax_Integer:7, OptionsPattern[]] := (
  numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
  AllowedNKSLTerms[#]]&/@Range[1, numEmax]] * 4;
  Print["Calculating " <> ToString[numValues] <> " values for Uk k
  =0,2,4,6."];
  counter = 1;
  If[And[OptionValue["Progress"], frontEndAvailable],
  progBar = PrintTemporary[
      Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
        counter}]]]
```

```
613       ];
614     ReducedUkTable = Table[
615       (
616         counter = counter+1;
617         {numE, 3, SL, SpLp, k} -> SimplifyFun[ReducedUk[numE, 3, SL,
      SpLp, k]]
618       ),
619       {numE, 1, numEmax},
620       {SL,   AllowedNKSLTerms[numE]},
621       {SpLp, AllowedNKSLTerms[numE]},
622       {k, {0, 2, 4, 6}}
623     ];
624     ReducedUkTable = Association[Flatten[ReducedUkTable]];
625     ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedUkTable.m"}];
626     If[And[OptionValue["Progress"], frontEndAvailable],
627       NotebookDelete[progBar]
628     ];
629     If[OptionValue["Export"],
630       (
631         Print["Exporting to file " <> ToString[ReducedUkTableFname]];
632         Export[ReducedUkTableFname, ReducedUkTable];
633       )
634     ];
635     Return[ReducedUkTable];
636   )
637
638   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.";
639   Options[GenerateReducedV1kTable] = {"Export" -> True, "Progress" ->
       True};
640   GenerateReducedV1kTable[numEmax_Integer:7, OptionsPattern[]] := (
641     numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
      AllowedNKSLTerms[#]]&/@Range[1, numEmax]];
642     Print["Calculating " <> ToString[numValues] <> " values for Vk1."
      ];
643     counter = 1;
644     If[And[OptionValue["Progress"], frontEndAvailable],
645     progBar = PrintTemporary[
646         Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
647           counter}]]]
648       ];
649     ReducedV1kTable = Table[
650       (
651         counter = counter+1;
652         {n, SL, SpLp, 1} -> SimplifyFun[ReducedV1k[n, SL, SpLp, 1]]
653       ),
654       {n, 1, numEmax},
655       {SL, AllowedNKSLTerms[n]},
656       {SpLp, AllowedNKSLTerms[n]}
657     ];
658     ReducedV1kTable = Association[ReducedV1kTable];
659     If[And[OptionValue["Progress"], frontEndAvailable],
660       NotebookDelete[progBar]
661     ];
662     exportFname = FileNameJoin[{moduleDir, "data", "ReducedV1kTable.m
      "}];
663     If[OptionValue["Export"],
664       (
665         Print["Exporting to file "<>ToString[exportFname]];
666         Export[exportFname, ReducedV1kTable];
667       )
668     ];
669     Return[ReducedV1kTable];
670   )
671
672   (* ###################### Racah Algebra ###################### *)
673   (* ########################################################## *)
674
675   (* ########################################################## *)
676   (* ###################### Electrostatic ###################### *)
677
678   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.";
679 fsubk[numE_, orbital_, NKSL_, NKSLp_, k_] := Module[
680   {terms, S, L, Sp, Lp,
681   termsWithSameSpin, SL,
682   fsubkVal, spinMultiplicity,
683   prefactor, summand1, summand2},
684   (
685     {S, L}   = FindSL[NKSL];
686     {Sp, Lp} = FindSL[NKSLp];
687     terms    = AllowedNKSLTerms[numE];
688     (* sum for summand1 is over terms with same spin *)
689     spinMultiplicity  = 2*S + 1;
690     termsWithSameSpin = StringCases[terms, ToString[
    spinMultiplicity] ~~ __];
691     termsWithSameSpin = Flatten[termsWithSameSpin];
692     If[Not[{S, L} == {Sp, Lp}],
693       Return[0]
694     ];
695     prefactor = 1/2 * Abs[Ck[orbital, k]]^2;
696     summand1 = Sum[(
697         ReducedUkTable[{numE, orbital, SL, NKSL,  k}] *
698         ReducedUkTable[{numE, orbital, SL, NKSLp, k}]
699         ),
700       {SL, termsWithSameSpin}
701     ];
702     summand1 = 1 / TPO[L] * summand1;
703     summand2 = (
704       KroneckerDelta[NKSL, NKSLp] *
705         (numE *(4*orbital + 2 - numE)) /
706         ((2*orbital + 1) * (4*orbital + 1))
707       );
708     fsubkVal = prefactor*(summand1 - summand2);
709     Return[fsubkVal];
710   )
711 ];
712
713 fsupk::usage = "fsupk[numE, orbital, SL, SLp, k] gives the
    superscripted Slater integral f^k = Subscript[f, k] * Subscript[D,
     k].";
714 fsupk[numE_, orbital_, NKSL_, NKSLp_ ,k_] := (
715   Dk[k] * fsubk[numE, orbital, NKSL, NKSLp, k]
716   )
717
718 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.";
719 Dk[k_] := {1, 225, 1089, 184041/25}[[k/2+1]];
720
721 FtoE::usage = "FtoE[F0, F2, F4, F6] calculates the Racah parameters
     {E0, E1, E2, E3} corresponding to the given Slater integrals.
722 See eqn. 2-80 in Wybourne.
723 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.";
724 FtoE[F0_, F2_, F4_, F6_] := Module[
725   {E0, E1, E2, E3},
726   (
727     E0 = (F0 - 10*F2/Dk[2] - 33*F4/Dk[4] - 286*F6/Dk[6]);
728     E1 = (70*F2/Dk[2] + 231*F4/Dk[4] + 2002*F6/Dk[6])/9;
729     E2 = (F2/Dk[2] - 3*F4/Dk[4] + 7*F6/Dk[6])/9;
730     E3 = (5*F2/Dk[2] + 6*F4/Dk[4] - 91*F6/Dk[6])/3;
731     Return[{E0, E1, E2, E3}];
732   )
733 ];
734
735 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.";
736 EtoF[E0_, E1_, E2_, E3_] := Module[
737   {F0, F2, F4, F6},
738   (
739     F0 = 1/7      (7 E0 + 9 E1);
```

```
740       F2 = 75/14    (E1 + 143 E2 + 11 E3);
741       F4 = 99/7     (E1 - 130 E2 + 4 E3);
742       F6 = 5577/350 (E1 + 35 E2 - 7 E3);
743       Return[{F0, F2, F4, F6}];
744     )
745   ];
746
747   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\".";
748   Options[Electrostatic] = {"Coefficients" -> "Slater"};
749   Electrostatic[{numE_, NKSL_, NKSLp_}, OptionsPattern[]] := Module[
750     {fsub0, fsub2, fsub4, fsub6,
751      esub0, esub1, esub2, esub3,
752      fsup0, fsup2, fsup4, fsup6,
753      eMatrixVal, orbital},
754     (
755       orbital = 3;
756       Which[
757         OptionValue["Coefficients"] == "Slater",
758         (
759           fsub0 = fsubk[numE, orbital, NKSL, NKSLp, 0];
760           fsub2 = fsubk[numE, orbital, NKSL, NKSLp, 2];
761           fsub4 = fsubk[numE, orbital, NKSL, NKSLp, 4];
762           fsub6 = fsubk[numE, orbital, NKSL, NKSLp, 6];
763           eMatrixVal = fsub0*F0 + fsub2*F2 + fsub4*F4 + fsub6*F6;
764         ),
765         OptionValue["Coefficients"] == "Racah",
766         (
767           fsup0 = fsupk[numE, orbital, NKSL, NKSLp, 0];
768           fsup2 = fsupk[numE, orbital, NKSL, NKSLp, 2];
769           fsup4 = fsupk[numE, orbital, NKSL, NKSLp, 4];
770           fsup6 = fsupk[numE, orbital, NKSL, NKSLp, 6];
771           esub0 = fsup0;
772           esub1 = 9/7*fsup0 +   1/42*fsup2 +   1/77*fsup4 +  1/462*
      fsup6;
773           esub2 =             143/42*fsup2 - 130/77*fsup4 + 35/462*
      fsup6;
774           esub3 =              11/42*fsup2 + 4/77*fsup4   -  7/462*
      fsup6;
775           eMatrixVal = esub0*E0 + esub1*E1 + esub2*E2 + esub3*E3;
776         )
777       ];
778       Return[eMatrixVal];
779     )
780   ];
781
782   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.";
783   Options[GenerateElectrostaticTable] = {"Export" -> True, "
        Coefficients" -> "Slater"};
784   GenerateElectrostaticTable[numEmax_Integer:7, OptionsPattern[]] :=
        (
785     ElectrostaticTable = Table[
786       {numE, SL, SpLp} -> SimplifyFun[Electrostatic[{numE, SL, SpLp},
       "Coefficients" -> OptionValue["Coefficients"]]],
787       {numE, 1, numEmax},
788       {SL, AllowedNKSLTerms[numE]},
789       {SpLp, AllowedNKSLTerms[numE]}
790     ];
791     ElectrostaticTable = Association[Flatten[ElectrostaticTable]];
792     If[OptionValue["Export"],
793       Export[FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"
      }],
794       ElectrostaticTable];
795     ];
796     Return[ElectrostaticTable];
797   );
798
```

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

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

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}.";
BasisTableGenerator[numE_] := Module[
  {energyStatesTable, allowedJ, J, Jp},
  (
    energyStatesTable = <||>;
    allowedJ = AllowedJ[numE];
    Do[
    (
      energyStatesTable[{numE, J}] = EnergyStates[numE, J];
    ),
    {Jp, allowedJ},
    {J,  allowedJ}];
    Return[energyStatesTable]
  )
];

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

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

```
        )
      ];
      Return[basis];
    )
];

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

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

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

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

NKCFPPhase::usage = "Phase between conjugate coefficients of
  fractional parentage according to Nielson and Koster page viii.
  Note that there is a typo on there the expression for zeta should
  be (-1)^((v-1)/2) instead of (-1)^(v - 1/2).";
NKCFPPhase[parent_, parentS_, parentL_, daughterS_, daughterL_,
  parentSeniority_, daughterSeniority_] := Module[
  {spin, orbital, expo, phase},
  (
    {spin, orbital} = {1/2, 3};
    expo = (
        (parentS + parentL + daughterS + daughterL) -
        (orbital + spin)
    );
    phase = Phaser[-expo];
    If[parent == 2*orbital,
        phase = phase * Phaser[(daughterSeniority-1)/2]];
    Return[phase];
  )
];

Options[CFPExpander] = {"Export" -> True, "PhaseFunction" -> "NK"};
CFPExpander::usage = "Using the coefficients of fractional
  parentage up to f7 this function calculates them up to f14.
```

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

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

    complementaryCFPs[{14, "1S"}] = {"1S", {"2F",1}};
    extendedCFPs         = Join[CFP, complementaryCFPs];
```

```
982       If[OptionValue["Export"];,
983       (
984           exportFname = FileNameJoin[{moduleDir, "data", "
      CFPs_extended.m"}];
985           Print["Exporting to ", exportFname];
986           Export[exportFname, extendedCFPs];
987       )
988       ];
989       Return[extendedCFPs];
990   )
991   ];
992
993   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.
994   The data being parsed here is the file attachment B1F_ALL.TXT which
       comes from Velkov's thesis.";
995   Options[GenerateCFPTable] = {"Export" -> True};
996   GenerateCFPTable[OptionsPattern[]] := Module[
997     {rawText, rawLines, leadChar, configIndex, line, daughter,
998     lineParts, numberCode, parsedNumber, toAppend, CFPTablefname},
999     (
1000      CleanWhitespace[string_]     := StringReplace[string,
      RegularExpression["\\s+"]->" "];
1001      AddSpaceBeforeMinus[string_] := StringReplace[string,
      RegularExpression["(?<!\\s)-"]->" -"];
1002      ToIntegerOrString[list_]     := Map[If[StringMatchQ[#,
      NumberString], ToExpression[#], #] &, list];
1003      CFPTable      = ConstantArray[{},7];
1004      CFPTable[[1]] = {{"2F",{"1S",1}}};
1005
1006      (* Cleaning before processing is useful *)
1007      rawText  = Import[FileNameJoin[{moduleDir, "data", "B1F_ALL.TXT
      "}]];
1008      rawLines = StringTrim/@StringSplit[rawText,"\n"];
1009      rawLines = Select[rawLines,#!=""&];
1010      rawLines = CleanWhitespace/@rawLines;
1011      rawLines = AddSpaceBeforeMinus/@rawLines;
1012
1013      Do[(
1014        (* the first character can be used to identify the start of a
       block *)
1015        leadChar=StringTake[line,{1}];
1016        (* ..FN, N is at position 50 in that line *)
1017        If[leadChar=="[",
1018        (
1019          configIndex=ToExpression[StringTake[line,{50}]];
1020          Continue[];
1021        )
1022        ];
1023        (* Identify which daughter term is being listed *)
1024        If[StringContainsQ[line,"[DAUGHTER TERM]"],
1025          daughter=StringSplit[line,"["][[1]];
1026          CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{
      daughter}];
1027          Continue[];
1028        ];
1029        (* Once we get here we are already parsing a row with
      coefficient data *)
1030        lineParts    = StringSplit[line," "];
1031        parent       = lineParts[[1]];
1032        numberCode   = ToIntegerOrString[lineParts[[3;;]]];
1033        parsedNumber = SquarePrimeToNormal[numberCode];
1034        toAppend     = {parent,parsedNumber};
1035        CFPTable[[configIndex]][[-1]] = Append[CFPTable[[configIndex
      ]][[-1]], toAppend]
1036      ),
1037      {line,rawLines}];
1038      If[OptionValue["Export"],
1039        (
1040        CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"
      }];
1041        Export[CFPTablefname, CFPTable];
1042        )
1043      ];
```

```
    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, " "],
          (
            multiplierSymbol = StringSplit[header, " "][[1]];
            multiplierValue = multiFactorValues[multiplierSymbol];
            operatorSymbol = StringSplit[header, " "][[2]];
            oppyS = Append[oppyS, operatorSymbol];
          ),
          (
            multiplierValue = 1;
            operatorSymbol = header;
          )
```

```
1236          ];
1237          normalValues = 1/multiplierValue*col[[2 ;;]];
1238          Join[{operatorSymbol}, normalValues], {col, tab1[[3 ;;]]}
1239        ];
1240
1241      (*Create an association for the matrix elements in the f^3 config
          *)
1242      juddOperators = Association[];
1243      Do[(
1244        col      = normalTable[[colIndex]];
1245        opLabel  = col[[1]];
1246        opValues = col[[2 ;;]];
1247        opMatrix = AssociationThread[matrixKeys -> opValues];
1248        Do[(
1249          opMatrix[Reverse[mKey]] = opMatrix[mKey]
1250          ),
1251        {mKey, matrixKeys}
1252        ];
1253        juddOperators[{3, opLabel}] = opMatrix),
1254        {colIndex, 1, Length[normalTable]}
1255      ];
1256
1257      (* special case of t2 in f3 *)
1258      (* this is the same as getting the matrix elements from Judd 1966
           *)
1259      numE = 3;
1260      e3Op      = juddOperators[{3, "e_{3}"}];
1261      t2prime   = juddOperators[{3, "t_{2}^{'}"}];
1262      prefactor = 1/(70 Sqrt[2]);
1263      t2Op = (# -> (t2prime[#] + prefactor*e3Op[#])) & /@ Keys[t2prime
          ];
1264      t2Op = Association[t2Op];
1265      juddOperators[{3, "t_{2}"}] = t2Op;
1266
1267      (*Special case of t11 in f3*)
1268      t11 = juddOperators[{3, "t_{11}"}];
1269      eβprimeOp = juddOperators[{3, "e_{\\beta}^{'}"}];
1270      t11primeOp = (# -> (t11[#] + Sqrt[3/385] eβprimeOp[#])) & /@ Keys
          [t11];
1271      t11primeOp = Association[t11primeOp];
1272      juddOperators[{3, "t_{11}^{'}"}] = t11primeOp;
1273      If[OptionValue["Export"],
1274        (
1275          (*export them*)
1276          PrintTemporary["Exporting ..."];
1277          exportFname = FileNameJoin[{moduleDir, "data", "juddOperators
              .m"}];
1278          Export[exportFname, juddOperators];
1279        )
1280      ];
1281      Return[juddOperators];
1282    );
1283
1284    GenerateThreeBodyTables::usage = "This function generates the
        matrix elements for the three body operators using the
        coefficients of fractional parentage, including those beyond f^7."
        ;
1285    Options[GenerateThreeBodyTables] = {"Export" -> False};
1286    GenerateThreeBodyTables[nmax_Integer : 14, OptionsPattern[]] := (
1287      tiKeys = {"t_{2}", "t_{2}^{'}", "t_{3}", "t_{4}", "t_{6}", "t_{7}
          ",
1288        "t_{8}", "t_{11}", "t_{11}^{'}", "t_{12}", "t_{14}", "t_{15}",
1289        "t_{16}", "t_{17}", "t_{18}", "t_{19}"};
1290      TSymbolsAssoc = AssociationThread[tiKeys -> TSymbols];
1291      juddOperators = ParseJudd1984[];
1292      (* 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. *)
1293      op3MatrixElement[SL_, SpLp_, opSymbol_] := (
1294        jOP = juddOperators[{3, opSymbol}];
1295        key = {SL, SpLp};
1296        val = If[MemberQ[Keys[jOP], key],
1297          jOP[key],
1298          0];
1299        Return[val];
1300      );
```

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

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

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

ReducedT11inf2::usage = "ReducedT11inf2[SL, SpLp] returns the
  reduced matrix element of the scalar component of the double
  tensor T11 for the given SL terms SL, SpLp.
Data used here for m0, m2, m4 is from Table II of Judd, BR, HM
  Crosswhite, and Hannah Crosswhite. Intra-Atomic Magnetic
  Interactions for f Electrons. Physical Review 169, no. 1 (1968):
  130.
";
ReducedT11inf2[SL_, SpLp_] := Module[
  {T11inf2},
  (
    T11inf2 = <|
      {"1S", "3P"} -> 6 M0 + 2 M2 + 10/11 M4,
      {"3P", "3P"} -> -36 M0 - 72 M2 - 900/11 M4,
      {"3P", "1D"} -> -Sqrt[(2/15)] (27 M0 + 14 M2 + 115/11 M4),
      {"1D", "3F"} -> Sqrt[2/5] (23 M0 + 6 M2 - 195/11 M4),
      {"3F", "3F"} -> 2 Sqrt[14] (-15 M0 - M2 + 10/11 M4),
      {"3F", "1G"} -> Sqrt[11] (-6 M0 + 64/33 M2 - 1240/363 M4),
      {"1G", "3H"} -> Sqrt[2/5] (39 M0 - 728/33 M2 - 3175/363 M4),
      {"3H", "3H"} -> 8/Sqrt[55] (-132 M0 + 23 M2 + 130/11 M4),
      {"3H", "1I"} -> Sqrt[26] (-5 M0 - 30/11 M2 - 375/1573 M4)
      |>;
    Which[
      MemberQ[Keys[T11inf2],{SL,SpLp}],
        Return[T11inf2[{SL,SpLp}]],
      MemberQ[Keys[T11inf2],{SpLp,SL}],
        Return[T11inf2[{SpLp,SL}]],
      True,
        Return[0]
      ]
  )
];

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

ReducedSOOandECSOinf2::usage = "ReducedSOOandECSOinf2[SL, SpLp]
  returns the reduced matrix element corresponding to the operator (
  T11 + t11 - a13 * z13 / 6) for the terms {SL, SpLp}. This
```

```
    combination of operators corresponds to the spin-other-orbit plus
    ECSO interaction.
1501 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\".
1502 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.
1503 Rigurously speaking the Pk parameters here are subscripted. The
    conversion to superscripted parameters is performed elsewhere with
     the Prescaling replacement rules.
1504 ";
1505 ReducedSOOandECSOinf2[SL_, SpLp_]  := Module[
1506   {a13, z13, z13inf2, matElement, redSOOandECSOinf2},
1507   (
1508     a13 = (-33 M0 + 3 M2 + 15/11 M4 -
1509           6 P0 + 3/2 (35 P2 + 77 P4 + 143 P6));
1510     z13inf2 = <|
1511         {"1S","3P"} -> 2,
1512         {"3P","3P"} -> 1,
1513         {"3P","1D"} -> -Sqrt[(15/2)],
1514         {"1D","3F"} -> Sqrt[10],
1515         {"3F","3F"} -> Sqrt[14],
1516         {"3F","1G"} -> -Sqrt[11],
1517         {"1G","3H"} -> Sqrt[10],
1518         {"3H","3H"} -> Sqrt[55],
1519         {"3H","1I"} -> -Sqrt[(13/2)]
1520         |>;
1521     matElement = Which[
1522       MemberQ[Keys[z13inf2],{SL,SpLp}],
1523         z13inf2[{SL,SpLp}],
1524       MemberQ[Keys[z13inf2],{SpLp,SL}],
1525         z13inf2[{SpLp,SL}],
1526       True,
1527         0
1528     ];
1529     redSOOandECSOinf2 = (
1530         ReducedT11inf2[SL, SpLp] +
1531         Reducedt11inf2[SL, SpLp] -
1532         a13 / 6 * matElement
1533     );
1534     redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
1535     Return[redSOOandECSOinf2];
1536   )
1537 ];
1538
1539 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.
1540 ";
1541 ReducedSOOandECSOinfn[numE_, SL_, SpLp_] := Module[
1542   {spin, orbital, t, S, L, Sp, Lp,
1543   idx1, idx2, cfpSL, cfpSpLp, parentSL,
1544   Sb, Lb, Sbp, Lbp, parentSpLp, funval},
1545   (
1546     {spin, orbital} = {1/2, 3};
1547     {S, L}   = FindSL[SL];
1548     {Sp, Lp} = FindSL[SpLp];
1549     t = 1;
1550     cfpSL    = CFP[{numE, SL}];
1551     cfpSpLp  = CFP[{numE, SpLp}];
1552     funval   = Sum[
1553       (
1554         parentSL = cfpSL[[idx2, 1]];
1555         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.
  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]
      ]
    )
  ];

  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];
    )
  ];

  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,
```

```
1736            0,
1737            numE==2,
1738            SimplifyFun[ReducedT22inf2[SL, SpLp]],
1739            True,
1740            SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
1741          ];
1742        ),
1743      {numE, 1, nmax},
1744      {SL,   AllowedNKSLTerms[numE]},
1745      {SpLp, AllowedNKSLTerms[numE]}
1746      ];
1747      If[And[OptionValue["Progress"],frontEndAvailable],
1748        NotebookDelete[progBar]
1749      ];
1750      If[OptionValue["Export"],
1751        (
1752          fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"
      }];
1753          Export[fname, T22Table];
1754        )
1755      ];
1756      Return[T22Table];
1757    );
1758
1759    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.
1760   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.\"
1761   \".
1762   ";
1763    SpinSpin[numE_, SL_, SpLp_, J_] := Module[
1764      {S, L, Sp, Lp, α, val},
1765      (
1766        α = 2;
1767        {S, L} = FindSL[SL];
1768        {Sp, Lp} = FindSL[SpLp];
1769        val = (
1770                 Phaser[Sp + L + J] *
1771                 SixJay[{Sp, Lp, J}, {L, S, α}] *
1772                 T22Table[{numE, SL, SpLp}]
1773               );
1774        Return[val]
1775      )
1776    ];
1777
1778    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.";
1779    Options[GenerateSpinSpinTable] = {"Export"->False};
1780    GenerateSpinSpinTable[nmax_, OptionsPattern[]] :=
1781      (
1782        SpinSpinTable = <||>;
1783        PrintTemporary[Dynamic[numE]];
1784        Do[
1785          SpinSpinTable[{numE, SL, SpLp, J}] = (SpinSpin[numE, SL, SpLp
      , J]);,
1786          {numE, 1, nmax},
1787          {J, MinJ[numE], MaxJ[numE]},
1788          {SL,   First /@ AllowedNKSLforJTerms[numE, J]},
1789          {SpLp, First /@ AllowedNKSLforJTerms[numE, J]}
1790          ];
1791          If[OptionValue["Export"],
1792          (fname = FileNameJoin[{moduleDir, "data", "SpinSpinTable.m"}];
1793            Export[fname, SpinSpinTable];
```

```
        )
      ];
      Return[SpinSpinTable];
    );

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CrystalField::usage = "CrystalField[n, NKSL, J, M, NKSLp, Jp, Mp]
  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}
  ]
)

TotalCFIters::usage = "TotalIters[i, j] returns total number of
  function evaluations for calculating all the matrix elements for
  the \!\(\*SuperscriptBox[\(f\), \(i\)]\) to the \!\(\*
  SuperscriptBox[\(f\), \(j\)]\) configurations.";
TotalCFIters[i_, j_] := (
```

```
    numIters = {196, 8281, 132496, 1002001, 4008004, 9018009,
    11778624};
    Return[Total[numIters[[i ;; j]]]];
    )

GenerateCrystalFieldTable::usage = "GenerateCrystalFieldTable[{
  numEs}] computes the matrix values for the crystal field
  interaction for f^n configurations the given list of numE in
  numEs. The function calculates the association CrystalFieldTable
  with keys of the form {numE, NKSL, J, M, NKSLp, Jp, Mp}. If the
  option \"Export\" is set to True, then the result is exported to
  the data subfolder for the folder in which this package is in. If
  the option \"Progress\" is set to True then an interactive
  progress indicator is shown. If \"Compress\" is set to true the
  exported values are compressed when exporting.";
Options[GenerateCrystalFieldTable] = {"Export" -> False, "Progress"
   -> True, "Compress" -> True};
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}
    ]
  )

  Options[ParseBenelli2015] = {"Export"->False};
  ParseBenelli2015[OptionsPattern[]] := Module[
    {fname, crystalSym,
    crystalSymmetries, parseFun,
    chars, qk, groupName, family,
    groupNum, params},
  (
    fname      = FileNameJoin[{moduleDir,"data","
   benelli_and_gatteschi_table3p3.csv"}];
    crystalSym = Import[fname][[2;;33]];
    crystalSymmetries = <||>;
    parseFun[txt_] := (
      chars = Characters[txt];
      qk    = chars[[-2;;]];
      If[chars[[1]]=="R",
      (
        Return[{ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}]}]
      ),
      (
        If[qk[[1]]=="0",
          Return[{ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}]}]
        ];
        Return[{
          ToExpression@StringJoin[{"B",qk[[1]],qk[[2]]}],
          ToExpression@StringJoin[{"S",qk[[1]],qk[[2]]}]
        }]
      )]
    );
    Do[
    (
      groupNum  = Round@ToExpression@row[[1]];
      groupName = row[[2]];
      family    = row[[3]];
      params    = Select[row[[4;;]],#=!=""&];
      params    = parseFun/@params;
      params    = <|"BqkSqk"->Sort@Flatten[params],
        "aliases"->{groupNum},
        "constraints"->{}|>;
      If[MemberQ[{"T","Th","O","Td","Oh"},groupName],
        params["constraints"] = {
            {B44 ->  Sqrt[5/14] B04, B46 -> -Sqrt[7/2] B06},
            {B44 -> -Sqrt[5/14] B04, B46 ->  Sqrt[7/2] B06}
            }
      ];
      If[StringContainsQ[groupName,","],
      (
        {alias1, alias2}          = StringSplit[groupName,","];
        crystalSymmetries[alias1] = params;
        crystalSymmetries[alias1]["aliases"] = {groupNum, alias2};
        crystalSymmetries[alias2] = params;
        crystalSymmetries[alias2]["aliases"] = {groupNum, alias1};
      ),
      (
      crystalSymmetries[groupName] = params;
      )
      ]
```

```
    ),
    {row,crystalSym}];
    crystalSymmetries["source"] = "Benelli and Gatteschi, 2015,
    Introduction to Molecular Magnetism, table 3.3.";
    If[OptionValue["Export"],
      Export[FileNameJoin[{moduleDir,"data","
    crystalFieldFunctionalForms.m"}],crystalSymmetries];
    ];
    Return[crystalSymmetries];
  )
  ]

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

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

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

  This uses data from table 3.3 in Benelli and Gatteschi, 2015.";
  CrystalFieldForm[symmetryGroupString_] := (
    If[Not@ValueQ[crystalFieldFunctionalForms],
      crystalFieldFunctionalForms = Import[FileNameJoin[{moduleDir, "
    data", "crystalFieldFunctionalForms.m"}]];
    ];
    crystalFieldFunctionalForms[symmetryGroupString]
  )

  (* ###################### 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,
      \[Alpha] * L * (L + 1),
      0
    ]
  )

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

  CasimirG2::usage = "CasimirG2[SL, SpLp] returns LS reduced matrix
```

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

GSO7W::usage = "GSO7W is an association whose keys are labels for
  the irreducible representations of group R7 and whose values are
  the eigenvalues of the corresponding Casimir operator.
Reference: Wybourne, \"Spectroscopic Properties of Rare Earths\",
  table 2-7.";
GSO7W := Association[
  {
    "000" -> 0,
    "100" -> 3/5,
    "110" -> 5/5,
    "111" -> 6/5,
    "200" -> 7/5,
    "210" -> 9/5,
    "211" -> 10/5,
    "220" -> 12/5,
    "221" -> 13/5,
    "222" -> 15/5
  }
];

CasimirSO7::usage = "CasimirSO7[SL, SpLp] returns the LS reduced
  matrix element of the configuration interaction term corresponding
   to the Casimir operator of R7.";
CasimirSO7[{SL_, SpLp_}] := (
  Wlabel = FindNKLSTerm[SL][[1]][[3]];
  If[SL==SpLp,
    γ * GSO7W[Wlabel],
    0
  ]
)

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

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

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

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

```
Options[JJBlockMatrix] = {"Sparse"->True, "ChenDeltas"->False};
JJBlockMatrix[numE_, J_, Jp_, CFTable_, OptionsPattern[]] := Module
  [
   {NKSLJMs, NKSLJMps, NKSLJM, NKSLJMp,
   SLterm, SpLpterm,
   MJ, MJp,
   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]
  )
  ];

  EnergyStates::usage = "Alias for AllowedNKSLJMforJTerms. At some
    point may be used to redefine states used in basis.";
  EnergyStates[numE_, J_]:= AllowedNKSLJMforJTerms[numE, J];

  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.";
  Options[JJBlockMatrixFileName] = {"FilenameAppendix" -> ""};
  JJBlockMatrixFileName[numE_Integer, OptionsPattern[]] := (
    fileApp = OptionValue["FilenameAppendix"];
    fname = FileNameJoin[{moduleDir,
        "hams",
        StringJoin[{"f", ToString[numE], "_JJBlockMatrixTable",
    fileApp ,".m"}]}];
    Return[fname];
  );

  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.";
  Options[TabulateJJBlockMatrixTable] = {"Sparse"->True, "ChenDeltas"
```

```
    ->False};
2277 TabulateJJBlockMatrixTable[numE_, CFTable_, OptionsPattern[]] := (
2278   JJBlockMatrixTable = <||>;
2279   totalIterations = Length[AllowedJ[numE]]^2;
2280   template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
2281   template2 = StringTemplate["`remtime` min remaining"];
2282   template4 = StringTemplate["Time elapsed = `runtime` min"];
2283   numiter   = 0;
2284   startTime = Now;
2285   If[$FrontEnd =!= Null,
2286     (
2287       temp = PrintTemporary[
2288         Dynamic[
2289           Grid[
2290             {
2291               {template1[<|"numiter"->numiter, "totaliter"->
    totalIterations|>]},
2292               {template2[<|"remtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime)/(Max[1,numiter])*(totalIterations-
    numiter), "min"]], 0.1]|>]},
2293               {template4[<|"runtime"->Round[QuantityMagnitude[
    UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
2294               {ProgressIndicator[numiter, {1, totalIterations}]}
2295             }
2296           ]
2297       ]
2298       ];
2299     )
2300   ];
2301   Do[
2302     (
2303       JJBlockMatrixTable[{numE, J, Jp}] = JJBlockMatrix[numE, J, Jp
    , CFTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas" ->
    OptionValue["ChenDeltas"]];
2304       numiter += 1;
2305     ),
2306   {Jp, AllowedJ[numE]},
2307   {J, AllowedJ[numE]}
2308   ];
2309   If[$FrontEnd =!= Null,
2310     NotebookDelete[temp]
2311   ];
2312   Return[JJBlockMatrixTable];
2313 );
2314
2315 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'|>.";
2316 Options[TabulateManyJJBlockMatrixTables] = {"Overwrite"->False, "
    Sparse"->True, "ChenDeltas"->False, "FilenameAppendix"-> "", "
    Compressed" -> False};
2317 TabulateManyJJBlockMatrixTables[ns_, OptionsPattern[]] := (
2318   overwrite = OptionValue["Overwrite"];
2319   fNames = <||>;
2320   fileApp = OptionValue["FilenameAppendix"];
2321   ExportFun = If[OptionValue["Compressed"], ExportMZip, Export];
2322   Do[
2323     (
2324       CFdataFilename = FileNameJoin[{moduleDir, "data", "
    CrystalFieldTable_f"<>ToString[numE]<>".zip"}];
2325       PrintTemporary["Importing CrystalFieldTable from ",
    CFdataFilename, " ..."];
2326       CrystalFieldTable = ImportMZip[CFdataFilename];
2327
2328       PrintTemporary["#------- numE = ", numE, " -------#"];
2329       exportFname = JJBlockMatrixFileName[numE, "FilenameAppendix"
    -> fileApp];
2330       fNames[numE] = exportFname;
2331       If[FileExistsQ[exportFname] && Not[overwrite],
2332         Continue[]
```

```
2333        ];
2334        JJBlockMatrixTable = TabulateJJBlockMatrixTable[numE,
      CrystalFieldTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas"
      -> OptionValue["ChenDeltas"]];
2335        If[FileExistsQ[exportFname]&&overwrite,
2336          DeleteFile[exportFname]
2337        ];
2338        ExportFun[exportFname, JJBlockMatrixTable];
2339
2340        ClearAll[CrystalFieldTable];
2341      ),
2342    {numE, ns}
2343    ];
2344    Return[fNames];
2345  );
2346
2347  HamMatrixAssembly::usage = "HamMatrixAssembly[numE] returns the
      Hamiltonian matrix for the f^n_i configuration. The matrix is
      returned as a SparseArray.
2348  The function admits an optional parameter \"FilenameAppendix\"
      which can be used to modify the filename to which the resulting
      array is exported to.
2349  It also admits an optional parameter \"IncludeZeeman\" which can be
       used to include the Zeeman interaction. The default is False
2350  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.
2351  The option \"ReturnInBlocks\" can be used to return the matrix in
      block or flattened form. The default is to return it in flattened
      form.";
2352  Options[HamMatrixAssembly] = {
2353        "FilenameAppendix"->"",
2354        "IncludeZeeman"->False,
2355        "Set t2Switch"->True,
2356        "ReturnInBlocks"->False};
2357  HamMatrixAssembly[nf_, OptionsPattern[]] := Module[
2358    {numE, ii, jj, howManyJs, Js, blockHam},
2359    (
2360      (*########################################*)
2361      ImportFun = ImportMZip;
2362      (*########################################*)
2363      (*hole-particle equivalence enforcement*)
2364      numE = nf;
2365      allVars = {E0, E1, E2, E3, ζ, F0, F2, F4, F6, M0, M2, M4, T2,
      T2p,
2366        T3, T4, T6, T7, T8, P0, P2, P4, P6, gs,
2367        α, β, γ, B02, B04, B06, B12, B14, B16,
2368        B22, B24, B26, B34, B36, B44, B46, B56, B66, S12, S14, S16,
      S22,
2369        S24, S26, S34, S36, S44, S46, S56, S66, T11, T11p, T12, T14,
      T15, T16,
2370        T17, T18, T19, Bx, By, Bz};
2371      params0 = AssociationThread[allVars, allVars];
2372      If[nf > 7,
2373        (
2374          numE = 14 - nf;
2375          params = HoleElectronConjugation[params0];
2376          If[OptionValue["Set t2Switch"], params[t2Switch] = 0];
2377        ),
2378        params = params0;
2379        If[OptionValue["Set t2Switch"], params[t2Switch] = 1];
2380      ];
2381      (* Load symbolic expressions for LS,J,J' energy sub-matrices.
      *)
2382      emFname = JJBlockMatrixFileName[numE, "FilenameAppendix" ->
      OptionValue["FilenameAppendix"]];
2383      JJBlockMatrixTable = ImportFun[emFname];
2384      (*Patch together the entire matrix representation using J,J'
      blocks.*)
2385      PrintTemporary["Patching JJ blocks ..."];
2386      Js        = AllowedJ[numE];
2387      howManyJs = Length[Js];
2388      blockHam  = ConstantArray[0, {howManyJs, howManyJs}];
2389      Do[
2390        blockHam[[jj, ii]] = JJBlockMatrixTable[{numE, Js[[ii]], Js[[
      jj]]}];,
```

```
2391        {ii, 1, howManyJs},
2392        {jj, 1, howManyJs}
2393        ];
2394
2395        (* Once the block form is created flatten it *)
2396        If[Not[OptionValue["ReturnInBlocks"]],
2397          (blockHam = ArrayFlatten[blockHam];
2398          blockHam  = ReplaceInSparseArray[blockHam, params];
2399          ),
2400          (blockHam  = Map[ReplaceInSparseArray[#, params]&,blockHam
    ,{2}];)
2401        ];
2402
2403        If[OptionValue["IncludeZeeman"],
2404          (
2405            PrintTemporary["Including Zeeman terms ..."];
2406            {magx, magy, magz} = MagDipoleMatrixAssembly[numE, "
    ReturnInBlocks" -> OptionValue["ReturnInBlocks"]];
2407            blockHam += - teslaToKayser * (Bx * magx + By * magy + Bz *
     magz);
2408          )
2409        ];
2410        Return[blockHam];
2411      )
2412    ];
2413
2414    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.";
2415    Options[SimplerSymbolicHamMatrix]={
2416      "Export"->True,
2417      "PrependToFilename"->"",
2418      "EorF"->"F",
2419      "Overwrite" -> False,
2420      "Return" -> True,
2421      "Set t2Switch" -> False,
2422      "IncludeZeeman" -> False};
2423    SimplerSymbolicHamMatrix[numE_Integer, simplifier_List,
      OptionsPattern[]] := Module[
2424      {thisHam,fname, fnamemx},
2425      (
2426        If[Not[ValueQ[ElectrostaticTable]],
2427          LoadElectrostatic[]
2428          ];
2429        If[Not[ValueQ[SOOandECSOTable]],
2430          LoadSOOandECSO[]
2431        ];
2432        If[Not[ValueQ[SpinOrbitTable]],
2433          LoadSpinOrbit[]
2434        ];
2435        If[Not[ValueQ[SpinSpinTable]],
2436          LoadSpinSpin[]
2437        ];
2438        If[Not[ValueQ[ThreeBodyTable]],
2439          LoadThreeBody[]
2440        ];
2441
2442        fname = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".m"}];
2443        fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"SymbolicMatrix-f"<>ToString[numE]<>".mx"}];
2444        If[Or[FileExistsQ[fname], FileExistsQ[fnamemx]] && Not[
    OptionValue["Overwrite"]],
2445          (
2446            If[OptionValue["Return"],
2447              (
2448                Which[
```

```
2449                FileExistsQ[fnamemx],
2450                (
2451                  Print["File ",fnamemx," already exists, and option \"Overwrite\" is set to False, loading file ..."];
2452                  thisHam = Import[fnamemx];
2453                  Return[thisHam];
2454                ),
2455                FileExistsQ[fname],
2456                (
2457                  Print["File ",fname," already exists, and option \"Overwrite\" is set to False, loading file ..."];
2458                  thisHam = Import[fname];
2459                  Print["Exporting to file ",fnamemx, " for quicker loading."];
2460                  Export[fnamemx,thisHam];
2461                  Return[thisHam];
2462                )
2463              ]
2464            ),
2465            (
2466              Print["File ",fname," already exists, skipping ..."];
2467              Return[Null];
2468            )
2469          ]
2470        )
2471      ];
2472
2473      thisHam = HamMatrixAssembly[numE, "Set t2Switch" -> OptionValue["Set t2Switch"], "IncludeZeeman"->OptionValue["IncludeZeeman"]];
2474      thisHam = ReplaceInSparseArray[thisHam, simplifier];
2475      (* This removes zero entries from being included in the sparse array *)
2476      thisHam = SparseArray[thisHam];
2477      If[OptionValue["Export"],
2478      (
2479        Print["Exporting to file ",fname, " and to ", fnamemx];
2480        Export[fname,thisHam];
2481        Export[fnamemx,thisHam];
2482      )
2483      ];
2484      If[OptionValue["Return"],
2485        Return[thisHam],
2486        Return[Null]
2487      ];
2488    )
2489  ];
2490
2491  (* ###################### Block assembly ###################### *)
2492  (* ########################################################### *)
2493
2494  (* ########################################################### *)
2495  (* ##################### Level Description ##################### *)
2496
2497  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.";
2498  FreeHam[JJBlocks_List, numE_Integer] := Module[
2499    {Js, basisJ, pivot, freeHam, idx, J,
2500    thisJbasis, shrunkBasisPositions, theBlock},
2501    (
2502      Js       = AllowedJ[numE];
2503      basisJ   = BasisLSJMJ[numE, "AsAssociation" -> True];
2504      pivot    = If[OddQ[numE], 1/2, 0];
2505      freeHam = Table[(
2506        J           = Js[[idx]];
2507        theBlock    = JJBlocks[[idx]];
2508        thisJbasis = basisJ[J];
2509        (* find the basis vectors that end with pivot *)
2510        shrunkBasisPositions = Flatten[Position[thisJbasis, {_ ..., pivot}]];
2511        (* take only those rows and columns *)
2512        theBlock[[shrunkBasisPositions, shrunkBasisPositions]]
2513      ),
2514      {idx, 1, Length[Js]}
2515      ];
2516      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}]
  )
];


LevelSimplerSymbolicHamMatrix::usage = "
  LevelSimplerSymbolicHamMatrix[numE] is a variation of
  HamMatrixAssembly that returns the diagonal JJ Hamiltonian blocks
  applying a simplifier and with simplifications adequate for the
  level description. The keys of the given association correspond to
   the different values of J that are possible for f^numE, the
  values are sparse array that are meant to be interpreted in the
  basis provided by BasisLSJ.
The option \"Simplifier\" is a list of symbols that are set to zero
  . At a minimum this has to include the crystal field parameters.
  By default this includes everything except the Slater parameters
  Fk and the spin orbit coupling ζ.
The option \"Export\" controls whether the resulting association is
   saved to disk, the default is True and the resulting file is
  saved to the ./hams/ folder. A hash is appended to the filename
  that corresponds to the simplifier used in the resulting
  expression. If the option \"Overwrite\" is set to False then these
   files may be used to quickly retrieve a previously computed case.
   The file is saved both in .m and .mx format.
The option \"PrependToFilename\" can be used to append a string to
  the filename to which the function may export to.
The option \"Return\" can be used to choose whether the function
  returns the matrix or not.
The option \"Overwrite\" can be used to overwrite the file if it
  already exists.";
Options[LevelSimplerSymbolicHamMatrix] = {
  "Export" -> True,
  "PrependToFilename" -> "",
  "Overwrite" -> False,
  "Return" -> True,
  "Simplifier" -> Join[
    {F0,\[Sigma]SS},
    cfSymbols,
    TSymbols,
```

```
      casimirSymbols,
      pseudoMagneticSymbols,
      marvinSymbols,
      DeleteCases[magneticSymbols,ζ]
    ]
  };
  LevelSimplerSymbolicHamMatrix[numE_Integer, OptionsPattern[]] :=
    Module[
    {thisHamAssoc, Js, fname,
    fnamemx, hash, simplifier},
    (
      simplifier = (#->0)&/@Sort[OptionValue["Simplifier"]];
      hash       = Hash[simplifier];
      If[Not[ValueQ[ElectrostaticTable]],LoadElectrostatic[]];
      If[Not[ValueQ[SOOandECSOTable]],LoadSOOandECSO[]];
      If[Not[ValueQ[SpinOrbitTable]],LoadSpinOrbit[]];
      If[Not[ValueQ[SpinSpinTable]],LoadSpinSpin[]];
      If[Not[ValueQ[ThreeBodyTable]],LoadThreeBody[]];
      fname    = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".m"}];
      fnamemx = FileNameJoin[{moduleDir,"hams",OptionValue["
    PrependToFilename"]<>"Level-SymbolicMatrix-f"<>ToString[numE]<>"-"
    <>ToString[hash]<>".mx"}];
      If[Or[FileExistsQ[fname],FileExistsQ[fnamemx]]&&Not[OptionValue
    ["Overwrite"]],
      (
        If[OptionValue["Return"],
        (
          Which[FileExistsQ[fnamemx],
          (
            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]
```

```
      ];
    )
];

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

```
      Clear[simpleHam];
      (* Eigensolver *)
      PrintFun["> Diagonalizing the numerical Hamiltonian within each separate J-subspace ..."];
      startTime     = Now;
      eigensys      = Eigensystem /@ numHam;
      endTime       = Now;
      diagonalTime  = QuantityMagnitude[endTime-startTime,"Seconds"];
      allEnergies   = Flatten[First/@Values[eigensys]];
      groundEnergy  = Min[allEnergies];
      eigensys      = Map[Chop[{#[[1]]-groundEnergy,#[[2]]}]&,eigensys];
      eigensys      = Association@KeyValueMap[#1->{basis[#1],#2} &, eigensys];
      PrintFun[">> Diagonalization took ",diagonalTime," seconds."];
      If[OptionValue["Return as states"],
        (
          PrintFun["> Padding the eigenvectors to correspond to the level basis ..."];
          eigenVectors  = SparseArray @ BlockDiagonalMatrix[Values[#[[2, 2]] & /@ eigensys]];
          globalBasis   = Flatten[Values[basis], 1];
          eigenEnergies = Flatten[Values[#[[2, 1]] & /@ eigensys]];
          eigenJs       = Flatten[KeyValueMap[ConstantArray[#1, Length[#2[[2, 2]]]] &, eigensys]];
          states        = Transpose[{eigenEnergies, eigenJs, eigenVectors}];
          states        = SortBy[states, First];
          eigenVectors  = Last /@ states;
          LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[InputForm[#[[2]]]]) & /@ globalBasis;
          majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ eigenVectors;
          levelLabels           = LSJmultiplets[[majorComponentIndices]];
          Return[{globalBasis, levelLabels, states}];
        ),
        Return[{basis, eigensys}]
      ];
    )
  ];


  (* ################### Level Description ################### *)
  (* ######################################################### *)

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

  magOp = <||>;

  JJBlockMagDip::usage = "JJBlockMagDip[numE, J, Jp] returns an array for the LSJM matrix elements of the magnetic dipole operator between states with given J and Jp. The option \"Sparse\" can be used to return a sparse matrix. The default is to return a sparse matrix.
  See eqn 15.7 in TASS.
  Here it is provided in atomic units in which the Bohr magneton is 1/2.
  \[Mu] = -(1/2) (L + gs S)
  We are using the Racah convention for the reduced matrix elements in the Wigner-Eckart theorem. See TASS eqn 11.15.
  ";
  Options[JJBlockMagDip]={"Sparse"->True};
  JJBlockMagDip[numE_, braJ_, ketJ_, OptionsPattern[]] := Module[
    {braSLJs, ketSLJs,
    braSLJ,   ketSLJ,
    braSL,    ketSL,
    braS,     braL,
    ketS,     ketL,
    braMJ,    ketMJ,
    matValue, magMatrix,
    summand1, summand2,
    threejays},
    (
      braSLJs  = AllowedNKSLJMforJTerms[numE,braJ];
```

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

  Options[TabulateJJBlockMagDipTable]={"Sparse"->True};
  TabulateJJBlockMagDipTable[numE_,OptionsPattern[]] := (
    JJBlockMagDipTable=<||>;
    Js=AllowedJ[numE];
    Do[
      (
        JJBlockMagDipTable[{numE,braJ,ketJ}]=
          JJBlockMagDip[numE,braJ,ketJ,"Sparse"->OptionValue["Sparse"
  ]]
      ),
    {braJ, Js},
    {ketJ, Js}
    ];
    Return[JJBlockMagDipTable]
  );

  TabulateManyJJBlockMagDipTables::usage = "
    TabulateManyJJBlockMagDipTables[{n1, n2, ...}] calculates the
    tables of matrix elements for the requested f^n_i configurations.
    The function does not return the matrices themselves. It instead
    returns an association whose keys are numE and whose values are
    the filenames where the output of TabulateManyJJBlockMagDipTables
    was saved to. The output consists of an association whose keys are
     of the form {n, J, Jp} and whose values are rectangular arrays
    given the values of <|LSJMJa|H_dip|L'S'J'MJ'a'|>.";
  Options[TabulateManyJJBlockMagDipTables]={"FilenameAppendix"->","
    Overwrite"->False,"Compressed"->True};
  TabulateManyJJBlockMagDipTables[ns_,OptionsPattern[]] := (
    fnames=<||>;
    Do[
    (
      ExportFun=If[OptionValue["Compressed"],ExportMZip,Export];
      PrintTemporary["#------- numE = ",numE," -------#"];
      appendTo     = (OptionValue["FilenameAppendix"]<>"-magDip");
      exportFname  = JJBlockMatrixFileName[numE,"FilenameAppendix"->
```

```
    appendTo];
2807      fnames[numE] = exportFname;
2808      If[FileExistsQ[exportFname]&&Not[OptionValue["Overwrite"]],
2809        Continue[]
2810      ];
2811      JJBlockMatrixTable = TabulateJJBlockMagDipTable[numE];
2812      If[FileExistsQ[exportFname]&&OptionValue["Overwrite"],
2813        DeleteFile[exportFname]
2814      ];
2815      ExportFun[exportFname,  JJBlockMatrixTable];
2816    ),
2817    {numE,ns}
2818    ];
2819    Return[fnames];
2820  );
2821
2822  MagDipoleMatrixAssembly::usage = "MagDipoleMatrixAssembly[numE]
     returns the matrix representation of the operator - 1/2 (L + gs S)
      in the f^numE configuration. The function returns a list with
     three elements corresponding to the x,y,z components of this
     operator. The option \"FilenameAppendix\" can be used to append a
     string to the filename from which the function imports from in
     order to patch together the array. For numE beyond 7 the function
     returns the same as for the complementary configuration. The
     option \"ReturnInBlocks\" can be used to return the matrices in
     blocks. The default is to return the matrices in flattened form
     and as sparse array.";
2823  Options[MagDipoleMatrixAssembly]={
2824    "FilenameAppendix"->"",
2825    "ReturnInBlocks"->False};
2826  MagDipoleMatrixAssembly[nf_Integer, OptionsPattern[]] := Module[
2827    {ImportFun, numE, appendTo,
2828    emFname, JJBlockMagDipTable,
2829    Js, howManyJs, blockOp,
2830    rowIdx, colIdx},
2831    (
2832      ImportFun = ImportMZip;
2833      numE      = nf;
2834      numH      = 14 - numE;
2835      numE      = Min[numE,numH];
2836
2837      appendTo  = (OptionValue["FilenameAppendix"]<>"-magDip");
2838      emFname   = JJBlockMatrixFileName[numE,"FilenameAppendix"->
     appendTo];
2839      JJBlockMagDipTable = ImportFun[emFname];
2840
2841      Js        = AllowedJ[numE];
2842      howManyJs = Length[Js];
2843      blockOp   = ConstantArray[0,{howManyJs,howManyJs}];
2844      Do[
2845        blockOp[[rowIdx,colIdx]] = JJBlockMagDipTable[{numE,Js[[
     rowIdx]],Js[[colIdx]]}],
2846        {rowIdx,1,howManyJs},
2847        {colIdx,1,howManyJs}
2848      ];
2849      If[OptionValue["ReturnInBlocks"],
2850        (
2851          opMinus = Map[#[[1]]&, blockOp, {4}];
2852          opZero  = Map[#[[2]]&, blockOp, {4}];
2853          opPlus  = Map[#[[3]]&, blockOp, {4}];
2854          opX =   (opMinus - opPlus)/Sqrt[2];
2855          opY = I (opPlus + opMinus)/Sqrt[2];
2856          opZ = opZero;
2857        ),
2858          blockOp = ArrayFlatten[blockOp];
2859          opMinus = blockOp[[;; , ;; , 1]];
2860          opZero  = blockOp[[;; , ;; , 2]];
2861          opPlus  = blockOp[[;; , ;; , 3]];
2862          opX =   (opMinus - opPlus)/Sqrt[2];
2863          opY = I (opPlus + opMinus)/Sqrt[2];
2864          opZ = opZero;
2865      ];
2866      Return[{opX, opY, opZ}];
2867    )
2868  ];
2869
```

```
2870  MagDipLineStrength::usage = "MagDipLineStrength[theEigensys, numE]
        takes the eigensystem of an ion and the number numE of f-electrons
         that correspond to it and calculates the line strength array Stot
        .
2871  The option \"Units\" can be set to either \"SI\" (so that the units
         of the returned array are (A m^2)^2) or to \"Hartree\".
2872  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.
2873  The returned array should be interpreted in the eigenbasis of the
        Hamiltonian. As such the element Stot[[i,i]] corresponds to the
        line strength states between states |i> and |j>.";
2874  Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI"
        , "States" -> All};
2875  MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern
        []] := Module[
2876    {allEigenvecs, Sx, Sy, Sz, Stot, factor},
2877    (
2878      numE = Min[14-numE0, numE0];
2879      (*If not loaded then load it, *)
2880      If[Or[
2881          Not[MemberQ[Keys[magOp], numE]],
2882          OptionValue["Reload MagOp"]],
2883      (
2884        magOp[numE] = ReplaceInSparseArray[#, {gs->2}]& /@
        MagDipoleMatrixAssembly[numE];
2885      )
2886      ];
2887      allEigenvecs = Transpose[Last /@ theEigensys];
2888      Which[OptionValue["States"] === All,
2889        (
2890          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
        allEigenvecs) & /@ magOp[numE];
2891          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2892        ),
2893        IntegerQ[OptionValue["States"]],
2894        (
2895          singleState  = theEigensys[[OptionValue["States"],2]];
2896          {Sx, Sy, Sz} = (ConjugateTranspose[allEigenvecs].#.
        singleState) & /@ magOp[numE];
2897          Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
2898        )
2899      ];
2900      Which[
2901        OptionValue["Units"] == "SI",
2902          Return[4 \[Mu]B^2 * Stot],
2903        OptionValue["Units"] == "Hartree",
2904          Return[Stot],
2905        True,
2906        (
2907          Print["Invalid option for \"Units\". Options are \"SI\" and
         \"Hartree\"."];
2908          Abort[];
2909        )
2910      ];
2911    )
2912  ];
2913
2914  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.
2915  Based on table 7.3 of Thorne 1999, using g2=1.
2916  The energy unit assumed in eigenSys is kayser.
2917  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>.
2918  By default this assumes that the refractive index is unity, this
```

```
    may be changed by setting the option \"RefractiveIndex\" to the
    desired value.
2919  The option \"Lifetime\" can be used to return the reciprocal of the
     transition rates. The default is to return the transition rates."
    ;
2920  Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False, "
    RefractiveIndex"->1};
2921  MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]] :=
    Module[
2922    {AMD, Stot, eigenEnergies,
2923    transitionWaveLengthsInMeters, nRefractive},
2924    (
2925      nRefractive    = OptionValue["RefractiveIndex"];
2926      numE           = Min[14-numE0, numE0];
2927      Stot           = MagDipLineStrength[eigenSys, numE, "Units" ->
    OptionValue["Units"]];
2928      eigenEnergies = Chop[First/@eigenSys];
2929      energyDiffs    = Outer[Subtract, eigenEnergies, eigenEnergies];
2930      energyDiffs    = ReplaceDiagonal[energyDiffs, Indeterminate];
2931      (* Energies assumed in kayser.*)
2932      transitionWaveLengthsInMeters = 0.01/energyDiffs;
2933
2934      unitFactor     = Which[
2935      OptionValue["Units"]=="Hartree",
2936      (
2937        (* The bohrRadius factor in SI needed to convert the
    wavelengths which are assumed in m*)
2938        16 \[Pi]^3 (\[Mu]0Hartree /(3 hPlanckFine)) * bohrRadius^3
2939      ),
2940      OptionValue["Units"]=="SI",
2941      (
2942        16 \[Pi]^3 \[Mu]0/(3 hPlanck)
2943      ),
2944      True,
2945      (
2946        Print["Invalid option for \"Units\". Options are \"SI\" and \
    "Hartree\"."];
2947        Abort[];
2948      )
2949      ];
2950      AMD = unitFactor / transitionWaveLengthsInMeters^3 * Stot *
    nRefractive^3;
2951      Which[OptionValue["Lifetime"],
2952        Return[1/AMD],
2953        True,
2954        Return[AMD]
2955      ]
2956    )
2957  ];
2958
2959  GroundMagDipoleOscillatorStrength::usage = "
    GroundMagDipoleOscillatorStrength[eigenSys, numE] calculates the
    magnetic diople oscillator strengths between the ground state and
    the excited states as given by eigenSys.
2960  Based on equation 8 of Carnall 1965, removing the 2J+1 factor since
     this degeneracy has been removed by the crystal field.
2961  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.
2962  The energy unit assumed in eigenSys is Kayser.
2963  The oscillator strengths are dimensionless.
2964  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>.
2965  By default this assumes that the refractive index is unity, this
    may be changed by setting the option \"RefractiveIndex\" to the
    desired value.";
2966  Options[GroundMagDipoleOscillatorStrength]={"RefractiveIndex"->1};
2967  GroundMagDipoleOscillatorStrength[eigenSys_List, numE_Integer,
    OptionsPattern[]] := Module[
2968    {eigenEnergies, SMDGS, GSEnergy, energyDiffs,
2969    transitionWaveLengthsInMeters, unitFactor, nRefractive},
2970    (
2971      eigenEnergies    = First/@eigenSys;
2972      nRefractive      = OptionValue["RefractiveIndex"];
2973      SMDGS            = MagDipLineStrength[eigenSys,numE, "Units"->"
```

```
  SI", "States"->1];
2974      GSEnergy          = eigenSys[[1,1]];
2975      energyDiffs       = eigenEnergies-GSEnergy;
2976      energyDiffs[[1]] = Indeterminate;
2977      transitionWaveLengthsInMeters = 0.01/energyDiffs;
2978      unitFactor        = (8\[Pi]^2 me)/(3 hPlanck eCharge^2 cLight);
2979      fMDGS             = unitFactor / transitionWaveLengthsInMeters *
     SMDGS * nRefractive;
2980      Return[fMDGS];
2981    )
2982  ];
2983
2984  (* ##################### Optical Operators ##################### *)
2985  (* ############################################################ *)
2986
2987  (* ############################################################ *)
2988  (* ################### Printers and Labels ################### *)
2989
2990  PrintL::usage = "PrintL[L] give the string representation of a
      given angular momentum.";
2991  PrintL[L_] := If[StringQ[L], L, StringTake[specAlphabet, {L + 1}]]
2992
2993  FindSL::usage = "FindSL[LS] gives the spin and orbital angular
      momentum that corresponds to the provided string LS.";
2994  FindSL[SL_] := (
2995    FindSL[SL] =
2996    If[StringQ[SL],
2997      {
2998        (ToExpression[StringTake[SL, 1]]-1)/2,
2999        StringPosition[specAlphabet, StringTake[SL, {2}]][[1, 1]]-1
3000      },
3001      SL
3002    ]
3003  );
3004
3005  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.";
3006  PrintSLJ[SLJ_] := (
3007    RowBox[{
3008      SuperscriptBox[" ", 2 SLJ[[1]] + 1],
3009      SubscriptBox[PrintL[SLJ[[2]]], SLJ[[3]]]
3010      }
3011      ] // DisplayForm
3012    );
3013
3014  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."
      ;
3015  PrintSLJM[SLJM_] := (
3016    RowBox[{
3017      SuperscriptBox[" ", 2 SLJM[[1]] + 1],
3018      SubscriptBox[PrintL[SLJM[[2]]], {SLJM[[3]], SLJM[[4]]}]
3019      }
3020      ] // DisplayForm
3021    );
3022
3023  (* ################### Printers and Labels ################### *)
3024  (* ############################################################ *)
3025
3026  (* ############################################################ *)
3027  (* ###################### Term management ###################### *)
3028
3029  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.";
3030  AllowedSLTerms[numE_] := Map[FindSL[First[#]] &, CFPTerms[Min[numE,
      14-numE]]];
```

```
3031
3032  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.";
3033  AllowedNKSLTerms[numE_] := Map[First, CFPTerms[Min[numE, 14-numE
        ]]];
3034  AllowedNKSLTerms[0]  = {"1S"};
3035  AllowedNKSLTerms[14] = {"1S"};
3036
3037  MaxJ::usage = "MaxJ[numE] gives the maximum J = S+L that
        corresponds to the configuration f^numE.";
3038  MaxJ[numE_] := Max[Map[Total, AllowedSLTerms[Min[numE, 14-numE]]]];
3039
3040  MinJ::usage = "MinJ[numE] gives the minimum J = S+L that
        corresponds to the configuration f^numE.";
3041  MinJ[numE_] := Min[Map[Abs[Part[#, 1] - Part[#, 2]] &,
        AllowedSLTerms[Min[numE, 14-numE]]]]
3042
3043  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.";
3044  AllowedSLJTerms[numE_] := Module[
3045    {idx1, allowedSL, allowedSLJ},
3046    (
3047      allowedSL = AllowedSLTerms[numE];
3048      allowedSLJ = {};
3049      For[
3050        idx1 = 1,
3051        idx1 <= Length[allowedSL],
3052        termSL = allowedSL[[idx1]];
3053        termsSLJ =
3054          Table[
3055            {termSL[[1]], termSL[[2]], J},
3056          {J,Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
3057          ];
3058        allowedSLJ = Join[allowedSLJ, termsSLJ];
3059        idx1++
3060      ];
3061      SortBy[allowedSLJ, Last]
3062    )
3063  ];
3064
3065  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.";
3066  AllowedNKSLJTerms[numE_] := Module[
3067    {allowedSL, allowedNKSL, allowedSLJ, nn},
3068    (
3069      allowedNKSL = AllowedNKSLTerms[numE];
3070      allowedSL = AllowedSLTerms[numE];
3071      allowedSLJ = {};
3072      For[
3073        nn = 1,
3074        nn <= Length[allowedSL],
3075        (
3076          termSL = allowedSL[[nn]];
3077          termNKSL = allowedNKSL[[nn]];
3078          termsSLJ =
3079            Table[{termNKSL, J},
3080            {J, Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
3081            ];
3082          allowedSLJ = Join[allowedSLJ, termsSLJ];
3083          nn++
3084        )
3085      ];
3086      SortBy[allowedSLJ, Last]
3087    )
3088  ];
3089
3090  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.";
3091  AllowedNKSLforJTerms[numE_, J_] := Module[
3092    {allowedSL, allowedNKSL, allowedSLJ,
3093    nn, termSL, termNKSL, termsSLJ},
3094    (
3095      allowedNKSL = AllowedNKSLTerms[numE];
3096      allowedSL = AllowedSLTerms[numE];
3097      allowedSLJ = {};
3098      For[
3099        nn = 1,
3100        nn <= Length[allowedSL],
3101        (
3102          termSL = allowedSL[[nn]];
3103          termNKSL = allowedNKSL[[nn]];
3104          termsSLJ = If[Abs[termSL[[1]] - termSL[[2]]] <= J <= Total[
    termSL],
3105            {{termNKSL, J}},
3106            {}
3107            ];
3108          allowedSLJ = Join[allowedSLJ, termsSLJ];
3109          nn++
3110        )
3111      ];
3112      Return[allowedSLJ]
3113    )
3114  ];
3115
3116  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
    }.";
3117  AllowedSLJMTerms[numE_] := Module[
3118    {allowedSLJ, allowedSLJM,
3119    termSLJ, termsSLJM, nn},
3120    (
3121    allowedSLJ = AllowedSLJTerms[numE];
3122    allowedSLJM = {};
3123    For[
3124      nn = 1,
3125      nn <= Length[allowedSLJ],
3126      nn++,
3127      (
3128        termSLJ = allowedSLJ[[nn]];
3129        termsSLJM =
3130          Table[{termSLJ[[1]], termSLJ[[2]], termSLJ[[3]], M},
3131          {M, - termSLJ[[3]], termSLJ[[3]]}
3132          ];
3133        allowedSLJM = Join[allowedSLJM, termsSLJM];
3134      )
3135    ];
3136    Return[SortBy[allowedSLJM, Last]];
3137    )
3138  ];
3139
3140  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}.";
3141  AllowedNKSLJMforJMTerms[numE_, J_, MJ_] := Module[
3142    {allowedSL, allowedNKSL,
3143    allowedSLJM, nn},
3144    (
3145      allowedNKSL = AllowedNKSLTerms[numE];
3146      allowedSL   = AllowedSLTerms[numE];
3147      allowedSLJM = {};
3148      For[
3149        nn = 1,
3150        nn <= Length[allowedSL],
3151        termSL = allowedSL[[nn]];
3152        termNKSL = allowedNKSL[[nn]];
3153        termsSLJ = If[(Abs[termSL[[1]] - termSL[[2]]]
3154                        <= J
3155                        <= Total[termSL]
3156                        && (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}.";
FindNKLSTerm[SL_] := Module[
  {NKterms, n},
  (
    n = 7;
    NKterms = {{}};
    Map[
      If[! StringFreeQ[First[#], SL],
        If[ToExpression[Part[#, 2]] <= n,
          NKterms = Join[NKterms, {#}, 1]
        ]
      ] &,
    fnTermLabels
    ];
    NKterms = DeleteCases[NKterms, {}];
    NKterms
  )
];

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}.";
Options[ParseTermLabels] = {"Export" -> True};
```

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

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

LoadParameters::usage = "LoadParameters[ln] takes a string with the
   symbol the element of a trivalent lanthanide ion and returns
  model parameters for it. It is based on the data for LaF3. If the
  option \"Free Ion\" is set to True then the function sets all
  crystal field parameters to zero. Through the option \"gs\" it
  allows modyfing the electronic gyromagnetic ratio. For
  completeness this function also computes the E parameters using
  the F parameters quoted on Carnall.";
Options[LoadParameters] = {
    "Source"->"Carnall",
    "Free Ion"->False,
    "gs"->2.002319304386,
    "With Uncertainties"->False
    };
LoadParameters[Ln_String, OptionsPattern[]] := Module[
  {source, params, uncertain,
  uncertainKeys, uncertainRules},
  (
    If[Not[ValueQ[Carnall]],
      LoadCarnall[];
    ];
    source = OptionValue["Source"];
    params = Which[source=="Carnall",
             (Association[Carnall["data"][Ln]])
             ];
    (*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*)
3279      params[M4] = 0.31*params[M0]; (*See Carnall 1989,Table I,
caption,probably fixed based on HF values*)
3280      params[P0] = 0;
3281      params[P4] = 0.5*params[P2];  (*See Carnall 1989,Table I,
caption,probably fixed based on HF values*)
3282      params[P6] = 0.1*params[P2];  (*See Carnall 1989,Table I,
caption,probably fixed based on HF values*)
3283      params[gs] = OptionValue["gs"];
3284      {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[
F0], params[F2], params[F4], params[F6]];
3285      params[E0] = 0;
3286      If[
3287        Not[OptionValue["With Uncertainties"]],
3288        Return[params],
3289        (
3290          uncertain     = Association[Carnall["annotations"][Ln]];
3291          uncertainKeys = Keys[uncertain];
3292          uncertain     = If[# == "Not allowed to vary in fitting."
|| # == "Interpolated",
3293              0., #] & /@ uncertain;
3294          paramKeys = Keys[params];
3295          uncertainVals = Sort[Intersection[paramKeys, uncertainKeys
]] /. Association[uncertain];
3296          uncertainRules = MapThread[Rule, {Sort[uncertainKeys],
uncertainVals}];
3297          Which[
3298            MemberQ[{"Ce", "Yb"}, Ln],
3299            (
3300              subsetL = {F0};
3301              subsetR = {0};
3302            ),
3303            True,
3304            (
3305            subsetL = {F0, M2, M4, P0, P4, P6, E0, E1, E2, E3};
3306            subsetR = {0, M0*0.65, M0*0.31, 0, P2*0.5, P2*0.1,
3307              0,
3308              Sqrt[(196 F2^2)/164025 + (49 F4^2)/88209 + (122500 F6
^2)/134165889],
3309              Sqrt[F2^2/4100625 + F4^2/10673289 + (30625 F6^2)
/2743558264161],
3310              Sqrt[F2^2/18225 + (4 F4^2)/1185921 + (30625 F6^2)
/1803785841]};
3311            )
3312          ];
3313          uncertainRules = Join[uncertainRules, MapThread[Rule, {
subsetL,subsetR /. uncertainRules}]];
3314          uncertainRules = Association[uncertainRules];
3315          Which[
3316            Ln == "Eu",
3317            (
3318              uncertainRules[F4] = 12.121;
3319              uncertainRules[F6] = 15.872;
3320            ),
3321            Ln == "Gd",
3322            (
3323              uncertainRules[F4] = 12.07;
3324            ),
3325            Ln == "Tb",
3326            (
3327              uncertainRules[F4] = 41.006;
3328            )
3329          ];
3330          If[MemberQ[{"Eu","Gd","Tb"},Ln],
3331            (
3332              uncertainRules[E1] = Sqrt[(196 F2^2)/164025 + (49 F4^2)
/88209 + (122500 F6^2)/134165889] /. uncertainRules;
3333              uncertainRules[E2] = Sqrt[F2^2/4100625 + F4^2/10673289
+ (30625 F6^2)/2743558264161] /. uncertainRules;
3334              uncertainRules[E3] = Sqrt[F2^2/18225 + (4 F4^2)/1185921
 + (30625 F6^2)/1803785841] /. uncertainRules;
3335            )
3336          ];
3337          uncertainKeys = First /@ Normal[uncertainRules];
3338          fullParams = Association[MapThread[Rule, {uncertainKeys,
MapThread[Around, {uncertainKeys /. params, uncertainKeys /.
```

```
    uncertainRules}]}]];
3339          Return[Join[params, fullParams]]
3340        )
3341        ];
3342    )
3343  ];
3344
3345  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.";
3346  HoleElectronConjugation[params_] := Module[
3347    {newparams = params},
3348    (
3349      flipSignsOf = {ζ, T2, T3, T4, T6, T7, T8};
3350      flipSignsOf = Join[flipSignsOf, cfSymbols];
3351      flipped =
3352        Table[(flipper -> - newparams[flipper]),
3353        {flipper, flipSignsOf}
3354        ];
3355      nonflipped =
3356        Table[(flipper -> newparams[flipper]),
3357        {flipper, Complement[Keys[newparams], flipSignsOf]}
3358        ];
3359      flippedParams = Association[Join[nonflipped, flipped]];
3360      flippedParams = Select[flippedParams, FreeQ[#, Missing]&];
3361      Return[flippedParams];
3362    )
3363  ];
3364
3365  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.
3366  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.
3367  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\".
3368  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.
3369  Tha ordered basis in which this eigenvector is to be interpreted is
       the one corresponding to BasisLSJMJ[numE].
3370  The function admits the following options:
3371  \"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.
3372  \"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.
3373  \"Zeroes\" (list) : A list with symbols assumed to be zero.
3374  ";
3375  Options[IonSolver] = {
3376    "Include Spin-Spin" -> True,
3377    "Overwrite Hamiltonian" -> False,
3378    "Zeroes" -> {}
3379  };
3380  IonSolver[numE_Integer, params0_Association, host_String:"Ln",
      OptionsPattern[]] := Module[
3381    {ln, simplifier, simpleHam, numHam, eigensys,
3382    startTime, endTime, diagonalTime,
3383    params=params0, zeroSymbols},
3384    (
3385      ln = theLanthanides[[numE]];
3386
3387      (* This could be done when replacing values, but this produces
      smaller saved arrays. *)
```

```
3388        simplifier = (#-> 0) & /@ OptionValue["Zeroes"];
3389        simpleHam = SimplerSymbolicHamMatrix[numE,
3390          simplifier,
3391          "PrependToFilename" -> host,
3392          "Overwrite" -> OptionValue["Overwrite Hamiltonian"]
3393        ];
3394
3395        (* Note that we don't have to flip signs of parameters for fn
      beyond f7 since the matrix produced
3396        by SimplerSymbolicHamMatrix has already accounted for this. *)
3397
3398        (* Everything that is not given is set to zero *)
3399        params = ParamPad[params, "Print" -> True];
3400        PrintFun[params];
3401
3402        (* Enforce the override to the spin-spin contribution to the
      magnetic interactions *)
3403        params[\[Sigma]SS] = If[OptionValue["Include Spin-Spin"], 1,
      0];
3404
3405        (* Create the numeric hamiltonian *)
3406        numHam = ReplaceInSparseArray[simpleHam, params];
3407        Clear[simpleHam];
3408
3409        (* Eigensolver *)
3410        PrintFun["> Diagonalizing the numerical Hamiltonian ..."];
3411        startTime = Now;
3412        eigensys  = Eigensystem[numHam];
3413        endTime   = Now;
3414        diagonalTime = QuantityMagnitude[endTime - startTime, "Seconds"
      ];
3415        PrintFun[">> Diagonalization took ", diagonalTime, " seconds."
      ];
3416        eigensys = Chop[eigensys];
3417        eigensys = Transpose[eigensys];
3418
3419        (* Shift the baseline energy *)
3420        eigensys = ShiftedLevels[eigensys];
3421        (* Sort according to energy *)
3422        eigensys = SortBy[eigensys, First];
3423        Return[eigensys];
3424      )
3425    ];
3426
3427    ShiftedLevels::usage = "ShiftedLevels[eigenSys] takes a list of
        levels of the form
3428    {{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.";
3429    ShiftedLevels[originalLevels_] := Module[
3430      {groundEnergy, shifted},
3431      (
3432        groundEnergy = Sort[originalLevels][[1,1]];
3433        shifted      = Map[{#[[1]] - groundEnergy, #[[2]]} &,
      originalLevels];
3434        Return[shifted];
3435      )
3436    ];
3437
3438
3439    (* ############################################################ *)
3440    (* ############## Optical Transitions for Levels ############## *)
3441
3442    JuddOfeltUkSquared::usage = "JuddOfeltUkSquared[numE, params]
        calculates the matrix elements of the Uk operator in the level
        basis. These are calculated according to equation (7) in Carnall
        1965.
3443    The function returns a list with the following elements:
3444      - basis : A list with the allowed {SL, J} terms in the f^numE
        configuration. Equal to BasisLSJ[numE].
3445      - eigenSys : A list with the eigensystem of the Hamiltonian for
        the f^n configuration.
3446      - levelLabels : A list with the labels of the major components of
         the level eigenstates.
3447      - LevelUkSquared : An association with the squared matrix
        elements of the Uk operators in the level eigenbasis. The keys
```

```
    being {2, 4, 6} corresponding to the rank of the Uk operator. The
    basis in which the matrix elements are given is the one
    corresponding to the level eigenstates given in eigenSys and whose
     major SLJ components are given in levelLabels. The matrix is
    symmetric and given as a SymmetrizedArray.
3448    The function admits the following options:
3449    \"PrintFun\" : A function that will be used to print the progress
     of the calculations. The default is PrintTemporary.";
3450  Options[JuddOfeltUkSquared] = {"PrintFun" -> PrintTemporary};
3451  JuddOfeltUkSquared[numE_, params_, OptionsPattern[]] := Module[
3452    {eigenChanger, numEH, basis, eigenSys,
3453    Js, Ukmat, LevelUkSquared, kRank,
3454    S, L, Sp, Lp, J, Jp, phase,
3455    braTerm, ketTerm, levelLabels,
3456    eigenVecs, majorComponentIndices},
3457    (
3458      If[Not[ValueQ[ReducedUkTable]],
3459        LoadUk[]
3460      ];
3461      numEH = Min[numE, 14-numE];
3462      PrintFun = OptionValue["PrintFun"];
3463      PrintFun["> Calculating the levels for the given parameters ...
    "];
3464      {basis, eigenSys} = LevelSolver[numE, params];
3465      (* The change of basis matrix to the eigenstate basis *)
3466      eigenChanger = Transpose[Last /@ eigenSys];
3467      PrintFun["Calculating the matrix elements of Uk in the physical
     coupling basis ..."];
3468      LevelUkSquared = <||>;
3469      Do[(
3470        Ukmat = Table[(
3471            {S, L}   = FindSL[braTerm[[1]]];
3472            J        = braTerm[[2]];
3473            Jp       = ketTerm[[2]];
3474            {Sp, Lp} = FindSL[ketTerm[[1]]];
3475            phase    = Phaser[S + Lp + J + kRank];
3476            Simplify @ (
3477              phase *
3478              Sqrt[TPO[J]*TPO[Jp]] *
3479              SixJay[{J, Jp, kRank}, {Lp, L, S}] *
3480              ReducedUkTable[{numEH, 3, braTerm[[1]], ketTerm[[1]],
    kRank}]
3481              )
3482            ),
3483          {braTerm, basis},
3484          {ketTerm, basis}
3485        ];
3486        Ukmat = (Transpose[eigenChanger] . Ukmat . eigenChanger)^2;
3487        Ukmat = Chop@Ukmat;
3488        LevelUkSquared[kRank] = SymmetrizedArray[Ukmat, Dimensions[
    eigenChanger], Symmetric[{1, 2}]];
3489        ),
3490        {kRank, {2, 4, 6}}
3491      ];
3492      LSJmultiplets = (RemoveTrailingDigits[#[[1]]] <> ToString[
    InputForm[#[[2]]]]) & /@ basis;
3493      eigenVecs = Last /@ eigenSys;
3494      majorComponentIndices = Ordering[Abs[#]][[-1]] & /@ eigenVecs;
3495      levelLabels = LSJmultiplets[[majorComponentIndices]];
3496      Return[{basis, eigenSys, levelLabels, LevelUkSquared}];
3497    )
3498  ];
3499
3500  LevelElecDipoleOscillatorStrength::usage = "
      LevelElecDipoleOscillatorStrength[numE, levelParams,
      juddOfeltParams] uses Judd-Ofelt theory to estimate the forced
      electric dipole oscillator strengths ions whose level description
      is determined by levelParams.
3501  The third parameter juddOfeltParams is an asssociation with keys
      equal to the three Judd-Ofelt intensity parameters {\[CapitalOmega
      ]2, \[CapitalOmega]4, \[CapitalOmega]6} and corresponding values
      in cm^2.
3502  The local field correction implemented here corresponds to the one
      given by the virtual cavity model of Lorentz.
3503  The function returns a list with the following elements:
3504    - basis : A list with the allowed {SL, J} terms in the f^numE
```

```
    configuration. Equal to BasisLSJ[numE].
    - eigenSys : A list with the eigensystem of the Hamiltonian for the f^n configuration in the level description.
    - levelLabels : A list with the labels of the major components of the calculated levels.
    - oStrengthArray : A square array whose elements represent the oscillator strengths between levels such that the element oStrengthArray[[i,j]] is the oscillator strength between the levels |Subscript[\[Psi], i]> and |Subscript[\[Psi], j]>. In this array, the elements below the diagonal represent emission oscillator strengths, and elements above the diagonal represent absorption oscillator strengths.
  The function admits the following three options:
    \"PrintFun\" : A function that will be used to print the progress of the calculations. The default is PrintTemporary.
    \"RefractiveIndex\" : The refractive index of the medium where the transitions are taking place. This may be a number or a function. If a number then the oscillator strengths are calculated for assuming a wavelength-independent refractive index. If a function then the refractive indices are calculated accordingly to the wavelength of each transition (the function must admit a single argument equal to the wavelength in nm). The default is 1.
    \"LocalFieldCorrection\" : The local field correction to be used. The default is \"VirtualCavity\". The options are: \"VirtualCavity\" and \"EmptyCavity\".
  The equation implemented here is the one given in eqn. 29 from the review article of Hehlen (2013). See that same article for a discussion on the local field correction.
  ";
  Options[LevelElecDipoleOscillatorStrength]={
    "PrintFun"        -> PrintTemporary,
    "RefractiveIndex" -> 1,
    "LocalFieldCorrection" -> "VirtualCavity"
  };
  LevelElecDipoleOscillatorStrength[numE_, levelParams_Association, juddOfeltParams_Association, OptionsPattern[]] := Module[
    {PrintFun, basis, eigenSys, levelLabels,
    LevelUkSquared, eigenEnergies, energyDiffs,
    oStrengthArray, nRef, \[Chi], nRefs,
    \[Chi]OverN, groundLevel, const,
    transitionFrequencies, wavelengthsInNM,
    fieldCorrectionType},
    (
      PrintFun = OptionValue["PrintFun"];
      nRef     = OptionValue["RefractiveIndex"];
      PrintFun["Calculating the Uk^2 matrix elements for the given parameters ..."];
      {basis, eigenSys, levelLabels, LevelUkSquared} = JuddOfeltUkSquared[numE, levelParams, "PrintFun"->PrintFun];
      eigenEnergies = First/@eigenSys;
      const         = (8\[Pi]^2)/3 me/hPlanck;
      energyDiffs   = Transpose@Outer[Subtract,eigenEnergies, eigenEnergies];
      (* since energies are assumed in Kayser, speed of light needs to be in cm/s, so that the frequencies are in 1/s *)
      transitionFrequencies = energyDiffs*cLight*100;
      (* grab the J for each level *)
      levelJs        = #[[2]] & /@ eigenSys;
      oStrengthArray = (
        juddOfeltParams[\[CapitalOmega]2]*LevelUkSquared[2]+
        juddOfeltParams[\[CapitalOmega]4]*LevelUkSquared[4]+
        juddOfeltParams[\[CapitalOmega]6]*LevelUkSquared[6]
      );
      oStrengthArray = Abs@(const * transitionFrequencies * oStrengthArray);
      (* it is necessary to divide each oscillator strength by the degeneracy of the initial level *)
      oStrengthArray = MapIndexed[1/(2 levelJs[[#2[[1]]]]+1) #1 &, oStrengthArray,{2}];
      (* including the effects of the refractive index *)
      fieldCorrectionType = OptionValue["LocalFieldCorrection"];
      Which[
        nRef === 1,
          True,
        NumberQ[nRef],
        (
```

```
3553            \[Chi] = Which[
3554              fieldCorrectionType == "VirtualCavity",
3555              (
3556                ( (nRef^2 + 2) / 3 )^2
3557              ),
3558              fieldCorrectionType == "EmptyCavity",
3559              (
3560                ( 3 * nRef^2 / ( 2 * nRef^2 + 1 ) )^2
3561              )
3562            ];
3563            \[Chi]OverN    = \[Chi] / nRef;
3564            oStrengthArray = \[Chi]OverN * oStrengthArray;
3565            (* the refractive index participates different in
       absorption and in emission *)
3566            aFunction      = If[#2[[1]]>#2[[2]], #1 * nRef^2, #1]&;
3567            oStrengthArray = MapIndexed[aFunction, oStrengthArray,
       {2}];
3568          ),
3569          True,
3570          (
3571            wavelengthsInNM = Abs[1 / energyDiffs] * 10^7;
3572            nRefs           = Map[nRef, wavelengthsInNM];
3573            Print["Calculating the oscillator strengths for the given
       refractive index ..."];
3574            \[Chi] = Which[
3575              fieldCorrectionType == "VirtualCavity",
3576              (
3577                ( (nRefs^2 + 2) / 3 )^2
3578              ),
3579              fieldCorrectionType == "EmptyCavity",
3580              (
3581                ( 3 * nRefs^2 / ( 2*nRefs^2 + 1 ) )^2
3582              )
3583            ];
3584            \[Chi]OverN     = \[Chi] / nRefs;
3585            oStrengthArray  = \[Chi]OverN * oStrengthArray
3586          )
3587        ];
3588        Return[{basis, eigenSys, levelLabels, oStrengthArray}];
3589      )
3590    ];
3591
3592    LevelJJBlockMagDipole::usage = "LevelJJBlockMagDipole[numE, J, Jp]
         returns an array of the LSJ reduced matrix elements of the
         magnetic dipole operator between states with given J and Jp. The
         option \"Sparse\" can be used to return a sparse matrix. The
         default is to return a sparse matrix.";
3593    Options[LevelJJBlockMagDipole] = {"Sparse"->True};
3594    LevelJJBlockMagDipole[numE_, braJ_, ketJ_, OptionsPattern[]] :=
         Module[
3595      {
3596        braSLJs, ketSLJs,
3597        braSLJ, ketSLJ,
3598        braSL, ketSL,
3599        braS, braL,
3600        ketS, ketL,
3601        matValue, magMatrix,
3602        summand1, summand2
3603      },
3604    (
3605      braSLJs     = AllowedNKSLforJTerms[numE,braJ];
3606      ketSLJs     = AllowedNKSLforJTerms[numE,ketJ];
3607      magMatrix   = Table[
3608        (
3609          braSL       = braSLJ[[1]];
3610          ketSL       = ketSLJ[[1]];
3611          {braS,braL} = FindSL[braSL];
3612          {ketS,ketL} = FindSL[ketSL];
3613          summand1    = If[Or[braJ!=ketJ,braSL!=ketSL],
3614            0,
3615            Sqrt[braJ*(braJ+1)*TPO[braJ]
3616            ]
3617          ];
3618          (*looking at the string includes checking L=L',S=S',and\alpha
       =\alpha'*)
3619          summand2    = If[braSL!=ketSL,
```

```
3620            0,
3621            (gs-1)*
3622            Phaser[braS+braL+ketJ+1]*
3623            Sqrt[TPO[braJ]*TPO[ketJ]]*
3624            SixJay[{braJ,1,ketJ},{braS,braL,braS}]*
3625            Sqrt[braS(braS+1)TPO[braS]]
3626          ];
3627          matValue = summand1 + summand2;
3628          matValue = -1/2 * matValue;
3629          matValue
3630        ),
3631        {braSLJ,braSLJs},
3632        {ketSLJ,ketSLJs}
3633      ];
3634      If[OptionValue["Sparse"],
3635        magMatrix = SparseArray[magMatrix]];
3636      Return[magMatrix];
3637    )
3638    ];
3639
3640    LevelMagDipoleMatrixAssembly::usage = "LevelMagDipoleMatrixAssembly
       [numE] puts together an array with the reduced matrix elements of
       the magnetic dipole operator in the level basis for the f^numE
       configuration. The function admits the two following options:
3641     \"Flattened\": If True then the returned matrix is flattened. The
        default is True.
3642     \"gs\": The electronic gyromagnetic ratio. The default is 2.";
3643    Options[LevelMagDipoleMatrixAssembly] = {
3644      "Flattened" -> True,
3645      gs -> 2
3646    };
3647    LevelMagDipoleMatrixAssembly[numE_, OptionsPattern[]] := Module[
3648      {Js, magDip, braJ, ketJ},
3649      (
3650        Js      = AllowedJ[numE];
3651        magDip  = Table[
3652          ReplaceInSparseArray[LevelJJBlockMagDipole[numE, braJ, ketJ],
       {gs -> OptionValue[gs]}],
3653          {braJ,Js},
3654          {ketJ,Js}
3655        ];
3656        If[OptionValue["Flattened"],
3657          magDip = ArrayFlatten[magDip];
3658        ];
3659        Return[magDip];
3660      )
3661    ];
3662
3663    LevelMagDipoleLineStrength::usage = "LevelMagDipoleLineStrength[
       eigenSys, numE] calculates the magnetic dipole line strengths for
       an ion whose level description is determined by levelParams. The
       function returns a square array whose elements represent the
       magnetic dipole line strengths between the levels given in
       eigenSys such that the element magDipoleLineStrength[[i,j]] is the
        line strength between the levels |Subscript[\[Psi], i]> and |
       Subscript[\[Psi], j]>. Eigensys must be such that it consists of a
        lists of lists where in each list the last element corresponds to
        the eigenvector of a level (given as a row) in the standard basis
        for levels of the f^numE configuration.
3664    The function admits the following options:
3665      \"Units\" : The units in which the line strengths are given. The
       default is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI
       \" then the unit of the line strength is (A m^2)^2 = (J/T)^2. If \
       "Hartree\" then the line strength is given in units of 2 \[Mu]B.";
3666    Options[LevelMagDipoleLineStrength] = {
3667      "Units"->"SI"
3668      };
3669    LevelMagDipoleLineStrength[theEigensys_List, numE0_Integer,
       OptionsPattern[]] := Module[
3670      {numE, levelMagOp, allEigenvecs, magDipoleLineStrength, units},
3671      (
3672        numE          = Min[14-numE0,numE0];
3673        levelMagOp    = LevelMagDipoleMatrixAssembly[numE];
3674        allEigenvecs  = Transpose[Last/@theEigensys];
3675        units         = OptionValue["Units"];
3676        magDipoleLineStrength          = Transpose[allEigenvecs].
```

```
  levelMagOp.allEigenvecs;
    magDipoleLineStrength         = Abs[magDipoleLineStrength]^2;
    Which[
      units=="SI",
        Return[4 \[Mu]B^2 * magDipoleLineStrength],
      units=="Hartree",
        Return[magDipoleLineStrength]
    ];
  )
];

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

```
3719            magDipoleOstrength = nRef * magDipoleOstrength;
3720          ),
3721        True,
3722          (
3723            wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
     10^7;
3724            nRefs              = Map[nRef, wavelengthsInNM];
3725            magDipoleOstrength = nRefs * magDipoleOstrength;
3726          )
3727      ];
3728      Return[{basis, eigenSys, magDipoleOstrength}];
3729    )
3730  ];
3731
3732  LevelMagDipoleSpontaneousDecayRates::usage = "
       LevelMagDipoleSpontaneousDecayRates[eigenSys, numE] calculates the
        spontaneous emission rates for the magnetic dipole transitions
       between the levels given in eigenSys. The function returns a
       square array whose elements represent the spontaneous emission
       rates between the levels given in eigenSys such that the element
       [[i,j]] of the returned array is the rate of spontaneous emission
       from the level |Subscript[\[Psi], i]> to the level |Subscript[\[
       Psi], j]>. In this array the elements below the diagonal represent
        emission rates, and elements above the diagonal are identically
       zero.
  The function admits two optional arguments:
3733    + \"Units\" : The units in which the rates are given. The default
3734     is \"SI\". The options are \"SI\" and \"Hartree\". If \"SI\" then
        the rates are given in s^-1. If \"Hartree\" then the rates are
       given in the atomic unit of frequency.
3735    + \"RefractiveIndex\" : The refractive index of the medium where
       the transitions are taking place. This may be a number or a
       function. If a number then the rates are calculated assuming a
       wavelength-independent refractive index as given. If a function
       then the refractive indices are calculated accordingly to the
       vaccum wavelength of each transition (the function must admit a
       single argument equal to the wavelength in nm). The default is 1."
       ;
3736  Options[LevelMagDipoleSpontaneousDecayRates] = {
3737    "Units" -> "SI",
3738    "RefractiveIndex" -> 1};
3739  LevelMagDipoleSpontaneousDecayRates[eigenSys_List, numE_Integer,
       OptionsPattern[]] := Module[
3740    {levMDlineStrength, eigenEnergies, energyDiffs,
3741    levelJs, spontaneousRatesInHartree, spontaneousRatesInSI,
3742    degenDivisor, units, nRef, nRefs, wavelengthsInNM},
3743    (
3744      nRef               = OptionValue["RefractiveIndex"];
3745      units              = OptionValue["Units"];
3746      levMDlineStrength =
       LowerTriangularize@LevelMagDipoleLineStrength[eigenSys,numE,"Units
       "->"Hartree"];
3747      levMDlineStrength = SparseArray[levMDlineStrength];
3748      eigenEnergies      = First /@ eigenSys;
3749      energyDiffs        = Outer[Subtract, eigenEnergies,
       eigenEnergies];
3750      energyDiffs        = kayserToHartree * energyDiffs;
3751      energyDiffs        = SparseArray[LowerTriangularize[energyDiffs
       ]];
3752      levelJs            = #[[2]]& /@ eigenSys;
3753      spontaneousRatesInHartree = 4/3 αFine^5 * energyDiffs^3 *
       levMDlineStrength;
3754      degenDivisor               = #1 / (2*levelJs[[#2[[1]]]] + 1)&;
3755      spontaneousRatesInHartree = MapIndexed[degenDivisor,
       spontaneousRatesInHartree, {2}];
3756      Which[nRef===1,
3757        True,
3758        NumberQ[nRef],
3759          (
3760            spontaneousRatesInHartree = nRef^3 *
       spontaneousRatesInHartree;
3761          ),
3762        True,
3763          (
3764            wavelengthsInNM    = Abs[kayserToHartree / energyDiffs] *
     10^7;
```

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

  (* ############# Optical Transitions for Levels ############## *)
  (* ############################################################ *)

  (* ############################################################ *)
  (* ################## Eigensystem analysis ################### *)

  PrettySaundersSLJmJ::usage = "PrettySaundersSLJmJ[{SL, J, mJ}]
    produces a human-redeable symbol for the given basis vector {SL, J
    , mJ}.";
  Options[PrettySaundersSLJmJ] = {"Representation" -> "Ket"};
  PrettySaundersSLJmJ[{SL_, J_, mJ_}, OptionsPattern[]] := (If[
    StringQ[SL],
    ({S, L} = FindSL[SL];
      L = StringTake[SL, {2, -1}];
      ),
    {S, L} = SL];
    pretty = RowBox[{AdjustmentBox[Style[2*S + 1, Smaller],
        BoxBaselineShift -> -1, BoxMargins -> 0],
        AdjustmentBox[PrintL[L], BoxMargins -> -0.2],
        AdjustmentBox[
        Style[{InputForm[J], mJ}, Small, FontTracking -> "Narrow"],
        BoxBaselineShift -> 1,
        BoxMargins -> {{0.7, 0}, {0.4, 0.4}}]}];
    pretty = DisplayForm[pretty];
    If[OptionValue["Representation"] == "Ket",
      pretty = Ket[pretty]
    ];
    Return[pretty];
  );

  PrettySaundersSLJ::usage = "PrettySaundersSLJ[{SL, J}] produces a
    human-redeable symbol for the given basis vector {SL, J}. SL can
    be either a list of two numbers representing S and L or a string
    representing the spin multiplicity and the total orbital angular
    momentum J in spectroscopic notation. The option \"Representation\
    " can be used to specify whether the output is given as a symbol
    or as a ket. The default is \"Ket\".";
  Options[PrettySaundersSLJ] = {"Representation"->"Ket"};
  PrettySaundersSLJ[{SL_,J_},OptionsPattern[]] := (
    If[StringQ[SL],
    (
      {S,L}=FindSL[SL];
      L=StringTake[SL,{2,-1}];
    ),
      {S,L}=SL
    ];
    pretty = RowBox[{
        AdjustmentBox[Style[2*S+1,Smaller],BoxBaselineShift->-1,
    BoxMargins->0],
        AdjustmentBox[PrintL[L],BoxMargins->-0.2],
        AdjustmentBox[Style[InputForm[J],Small,FontTracking->"Narrow"
    ],BoxBaselineShift->1,BoxMargins->{{0.7,0},{0.4,0.4}}]
      }
    ];
    pretty = DisplayForm[pretty];
    pretty = Which[
      OptionValue["Representation"]=="Ket",
        Ket[pretty],
      OptionValue["Representation"]=="Symbol",
        pretty
```

```
    ];
    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}
        ),
        {state, eigensys}];
      Return[parsedStates]
    )
  ];

  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.";
  FindThresholdPosition[list_, threshold_] := Module[
    {position},
    (
    position = Position[list, _?(# >= threshold &), 1, 1];
    thrPos = If[Length[position] > 0,
      position[[1, 1]],
      Length[list]];
    If[thrPos == 0,
      Return[1],
      Return[thrPos]]
    )
  ];

  ParseStateByProbabilitySum[{energy_, eigenVec_}, probSum_, roundTo_:0.01, maxParts_:20] := Compile[
    {{energy, _Real, 0}, {eigenVec, _Complex, 1},
     {probSum, _Real, 0},{roundTo, _Real, 0},
     {maxParts, _Integer, 0}},
      Module[
        {numStates, state, amplitudes, probs, ordering,
        orderedProbs, truncationIndex, accProb, thresholdIndex, chosenIndices, majorComponents,
        majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep},
        (
          numStates    = Length[eigenVec];
          (*Round them up*)
          amplitudes         = Round[eigenVec, roundTo];
          probs              = Round[Abs[eigenVec^2], roundTo];
          ordering           = Reverse[Ordering[probs]];
          (*Order the probabilities from high to low*)
```

```
3945            orderedProbs       = probs[[ordering]];
3946            (*To speed up Accumulate, assume that only as much as
    maxParts will be needed*)
3947            truncationIndex    = Min[maxParts, Length[orderedProbs]];
3948            orderedProbs       = orderedProbs[[;;truncationIndex]];
3949            (*Accumulate the probabilities*)
3950            accProb            = Accumulate[orderedProbs];
3951            (*Find the index of the first element in accProb that is
    greater than probSum*)
3952            thresholdIndex     = Min[Length[accProb],
    FindThresholdPosition[accProb, probSum]];
3953            (*Grab all the indicees up till that one*)
3954            chosenIndices      = ordering[[;; thresholdIndex]];
3955            (*Select the corresponding elements from the basis*)
3956            majorComponents    = basis[[chosenIndices]];
3957            (*Select the corresponding amplitudes*)
3958            majorAmplitudes    = amplitudes[[chosenIndices]];
3959            (*Take their absolute value*)
3960            absMajorAmplitudes = Abs[majorAmplitudes];
3961            (*Make sure that there are no effectively zero
    contributions*)
3962            notnullAmplitudes  = Flatten[Position[absMajorAmplitudes,
    x_ /; x != 0]];
3963            (* majorComponents    = PrettySaundersSLJmJ
    [{#[[1]],#[[2]],#[[3]]} & /@ majorComponents; *)
3964            majorComponents    = PrettySaundersSLJmJ /@ majorComponents
    ;
3965            majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
3966            (*Make them into Kets*)
3967            majorComponents    = Ket /@ majorComponents[[
    notnullAmplitudes]];
3968            (*Multiply and add to build the final Ket*)
3969            majorRep           = majorAmplitudes . majorComponents;
3970            Return[{energy, majorRep}];
3971        )
3972      ],
3973    CompilationTarget -> "C",
3974    RuntimeAttributes -> {Listable},
3975    Parallelization -> True,
3976    RuntimeOptions -> "Speed"
3977  ];
3978
3979  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.";
3980  ParseStatesByProbabilitySum[eigensys_, basis_, probSum_, roundTo_ :
      0.01, maxParts_: 20] := Module[
3981    {parsedByProb, numStates, state, energy,
3982    eigenVec, amplitudes, probs, ordering,
3983    orderedProbs, truncationIndex, accProb,
3984    thresholdIndex, chosenIndices, majorComponents,
3985    majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep
      },
3986    (
3987      numStates    = Length[eigensys];
3988      parsedByProb = Table[(
3989        state              = eigensys[[idx]];
3990        {energy, eigenVec} = state;
3991        (*Round them up*)
3992        amplitudes         = Round[eigenVec, roundTo];
3993        probs              = Round[Abs[eigenVec^2], roundTo];
3994        ordering           = Reverse[Ordering[probs]];
3995        (*Order the probabilities from high to low*)
3996        orderedProbs       = probs[[ordering]];
3997        (*To speed up Accumulate, assume that only as much as
    maxParts will be needed*)
3998        truncationIndex    = Min[maxParts, Length[orderedProbs]];
3999        orderedProbs       = orderedProbs[[;;truncationIndex]];
4000        (*Accumulate the probabilities*)
4001        accProb            = Accumulate[orderedProbs];
4002        (*Find the index of the first element in accProb that is
    greater than probSum*)
4003        thresholdIndex     = Min[Length[accProb],
    FindThresholdPosition[accProb, probSum]];
```

```
4004          (*Grab all the indicees up till that one*)
4005          chosenIndices      = ordering[[;; thresholdIndex]];
4006          (*Select the corresponding elements from the basis*)
4007          majorComponents    = basis[[chosenIndices]];
4008          (*Select the corresponding amplitudes*)
4009          majorAmplitudes    = amplitudes[[chosenIndices]];
4010          (*Take their absolute value*)
4011          absMajorAmplitudes = Abs[majorAmplitudes];
4012          (*Make sure that there are no effectively zero contributions
               *)
4013          notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_
             /; x != 0]];
4014          (* majorComponents    = PrettySaundersSLJmJ
             [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
4015          majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
4016          majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
4017          majorComponents    = majorComponents[[notnullAmplitudes]];
4018          (*Multiply and add to build the final Ket*)
4019          majorRep           = majorAmplitudes . majorComponents;
4020          {energy, majorRep}
4021          ), {idx, numStates}];
4022        Return[parsedByProb];
4023      )
4024    ];
4025
4026    (* ################## Eigensystem analysis ################### *)
4027    (* ############################################################ *)
4028
4029    (* ############################################################ *)
4030    (* ########################## Misc ############################ *)
4031
4032    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.";
4033    SymbToNum[expr_, replacementAssociation_] := (
4034      includedKeys = Keys[replacementAssociation];
4035      (*If a key is not defined, make its value zero.*)
4036      fullAssociation = Table[(
4037        If[MemberQ[includedKeys, key],
4038          ToExpression[key]->replacementAssociation[key],
4039          ToExpression[key]->0
4040        ]
4041      ),
4042      {key, paramSymbols}];
4043      Return[expr/.fullAssociation];
4044    );
4045
4046    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.";
4047    SimpleConjugate[expr_] := expr /. Complex[a_, b_] :> a - I b;
4048
4049    ExportMZip::usage = "ExportMZip[\"dest.[zip,m]\"] saves a
        compressed version of expr to the given destination.";
4050    ExportMZip[filename_, expr_] := Module[
4051      {baseName, exportName, mImportName, zipImportName},
4052      (
4053        baseName    = FileBaseName[filename];
4054        exportName  = StringReplace[filename,".m"->".zip"];
4055        mImportName = StringReplace[exportName,".zip"->".m"];
4056        If[FileExistsQ[mImportName],
4057        (
4058          PrintTemporary[mImportName<>" exists already, deleting"];
4059          DeleteFile[mImportName];
4060          Pause[2];
4061        )
4062        ];
4063        Export[exportName, (baseName<>".m") -> expr];
4064      )
4065    ];
```

```
4066
4067  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";
4068  Options[ImportMZip]={"Leave Uncompressed" -> True};
4069  ImportMZip[filename_String, OptionsPattern[]] := Module[
4070    {baseName, importKey, zipImportName, mImportName, imported},
4071    (
4072      baseName      = FileBaseName[filename];
4073      (*Function allows for the filename to be .m or .zip*)
4074      importKey     = baseName <> ".m";
4075      zipImportName = StringReplace[filename, ".m"->".zip"];
4076      mImportName   = StringReplace[zipImportName, ".zip"->".m"];
4077      If[FileExistsQ[mImportName],
4078      (
4079        PrintTemporary[".m version exists already, importing that
        instead ..."];
4080        Return[Import[mImportName]];
4081      )
4082      ];
4083      imported = Import[zipImportName, importKey];
4084      If[OptionValue["Leave Uncompressed"],
4085        Export[mImportName, imported]
4086      ];
4087      Return[imported]
4088    )
4089  ];
4090
4091  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.";
4092  ReplaceInSparseArray[sparseA_SparseArray, rules_] := (
4093    SparseArray[Automatic,
4094      sparseA["Dimensions"],
4095      sparseA["Background"] /. rules,
4096      {
4097        1,
4098        {sparseA["RowPointers"], sparseA["ColumnIndices"]},
4099        sparseA["NonzeroValues"] /. rules
4100      }
4101      ]
4102  );
4103
4104  MapToSparseArray::usage = "MapToSparseArray[sparseArray, function]
        takes a sparse array and returns a sparse array after the function
         has been applied to it.";
4105  MapToSparseArray[sparseA_SparseArray, func_] := Module[
4106    {nonZ, backg, mapped},
4107    (
4108      nonZ   = func /@ sparseA["NonzeroValues"];
4109      backg  = func[sparseA["Background"]];
4110      mapped = SparseArray[Automatic,
4111        sparseA["Dimensions"],
4112        backg,
4113        {
4114          1,
4115          {sparseA["RowPointers"], sparseA["ColumnIndices"]},
4116          nonZ
4117        }
4118      ];
4119      Return[mapped];
4120    )
4121  ];
4122
4123  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.";
4124  Options[ParseTeXLikeSymbol] = {"Form" -> "List"};
4125  ParseTeXLikeSymbol[bigString_, OptionsPattern[]] := Module[
```

```
    {form, mainSymbol, symbols},
    (
      form = OptionValue["Form"];
      (* parse greek *)
      symbols = Table[(
          str = StringReplace[string, {"\\alpha" -> "α",
            "\\beta" -> "β",
            "\\gamma" -> "γ",
            "\\psi" -> "\[Psi]"}];
          symbol = Which[
            StringContainsQ[str, "_"] && Not[StringContainsQ[str, "^"]],
            (
              (*yes sub no sup*)
              mainSymbol = StringSplit[str, "_"][[1]];
              mainSymbol = ToExpression[mainSymbol];

              subPart =
                StringCases[str,
                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
              Subscript[mainSymbol, subPart]
            ),
            Not[StringContainsQ[str, "_"]] && StringContainsQ[str, "^"],
            (
              (*no sub yes sup*)
              mainSymbol = StringSplit[str, "^"][[1]];
              mainSymbol = ToExpression[mainSymbol];

              supPart =
                StringCases[str,
                  RegularExpression@"\\{(.*?)\\}" -> "$1"][[1]];
              Superscript[mainSymbol, supPart]
            ),
            StringContainsQ[str, "_"] && StringContainsQ[str, "^"],
            (
              (*yes sub yes sup*)
              mainSymbol = StringSplit[str, "_"][[1]];
              mainSymbol = ToExpression[mainSymbol];
              {subPart, supPart} =
                StringCases[str, RegularExpression@"\\{(.*?)\\}" -> "$1"];
              Subsuperscript[mainSymbol, subPart, supPart]
            ),
            True,
            (
              (*no sup or sub*)
              str
            )
            ];
          symbol
          ),
        {string, StringSplit[bigString, " "]}
      ];
      Which[
        form == "Row",
        Return[Row[symbols]],
        form == "List",
        Return[symbols]
      ]
    )
  ];

  FromArrayToTable::usage = "FromArrayToTable[array, labels, energies
    ] takes a square array of values and returns a table with the
    labels of the rows and columns, the energies of the initial and
    final levels, the level energies, the vacuum wavelength of the
    transition, and the value of the array. The array must be square
    and the labels and energies must be compatible with the order
    implied by the array. The array must be a square array of values.
    The function returs a list of lists with the following elements:
    - Initial level index
    - Final level index
    - Initial level label
    - Final level label
    - Initial level energy
```

```
4192      - Final level energy
4193      - Vacuum wavelength
4194      - Value of the array element.
4195    Elements in which the array is zero are not included in the return
          of this function.";
4196    FromArrayToTable[array_,labels_,energies_] := Module[
4197      {tableFun, atl},
4198      (
4199        tableFun = {
4200          #2[[1]],
4201          #2[[2]],
4202          labels[[#2[[1]]]],
4203          labels[[#2[[2]]]],
4204          energies[[#2[[1]]]],
4205          energies[[#2[[2]]]],
4206          If[#2[[1]]==#2[[2]],"--",10^7/(energies[[#2[[1]]]]-energies
          [[#2[[2]]]])],
4207          #1
4208        }&;
4209        atl = Select[Flatten[MapIndexed[tableFun,array
          ,{2}],1],#[[-1]]=!=0.&];
4210        atl = Append[#,1/#[[-1]]]&/@atl;
4211        Return[atl]
4212      )
4213    ]
4214    (* ########################## Misc ########################### *)
4215    (* ############################################################ *)
4216
4217    (* ############################################################ *)
4218    (* ################## Some Plotting Routines ################## *)
4219
4220    EnergyLevelDiagram::usage = "EnergyLevelDiagram[states] takes
          states and produces a visualization of its energy spectrum.
4221    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."
          ;
4222    Options[EnergyLevelDiagram] = {
4223      "Title"->"",
4224      "ImageSize"->1000,
4225      "AspectRatio" -> 1/8,
4226      "Background"->"Automatic",
4227      "Epilog"->{},
4228      "Explorer"->True
4229      };
4230    EnergyLevelDiagram[states_, OptionsPattern[]] := Module[
4231      {energies, epi, explora},
4232      (
4233        energies = First/@states;
4234        epi = OptionValue["Epilog"];
4235        explora = If[OptionValue["Explorer"],
4236          ExploreGraphics,
4237          Identity
4238        ];
4239        explora@ListPlot[Tooltip[{{#, 0}, {#, 1}}, {Quantity
          [#/8065.54429, "eV"], Quantity[#, 1/"Centimeters"]}] &/@ energies,
4240          Joined       -> True,
4241          PlotStyle    -> Black,
4242          AspectRatio  -> OptionValue["AspectRatio"],
4243          ImageSize    -> OptionValue["ImageSize"],
4244          Frame        -> True,
4245          PlotRange    -> {All, {0, 1}},
4246          FrameTicks   -> {{None, None}, {Automatic, Automatic}},
4247          FrameStyle   -> Directive[15, Dashed, Thin],
4248          PlotLabel    -> Style[OptionValue["Title"], 15, Bold],
4249          Background   -> OptionValue["Background"],
4250          FrameLabel   -> {"\!\(\*FractionBox[\(E\), SuperscriptBox[\(
          cm\), \(-1\)]]\)"},
4251          Epilog       -> epi]
4252      )
4253    ];
4254
4255    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.
4256    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};
        )}]]];

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.";
Options[LabeledGrid]={
    ItemSize->Automatic,
    Alignment->Center,
    Frame->All,
    "Separator"->",",
    "Pivot"->""
};
```

```
LabeledGrid[data_,rowHeaders_,columnHeaders_,OptionsPattern[]] :=
  Module[
   {gridList=data, rowHeads=rowHeaders, colHeads=columnHeaders},
   (
     separator=OptionValue["Separator"];
     pivot=OptionValue["Pivot"];
     gridList=Table[
             Tooltip[
               data[[rowIdx,colIdx]],
               DisplayForm[
                 RowBox[{rowHeads[[rowIdx]],
                         separator,
                         colHeads[[colIdx]]}
                       ]
                       ]
                     ],
         {rowIdx,Dimensions[data][[1]]},
         {colIdx,Dimensions[data][[2]]}];
     gridList=Transpose[Prepend[gridList,colHeads]];
     rowHeads=Prepend[rowHeads,pivot];
     gridList=Prepend[gridList,rowHeads]//Transpose;
     Grid[gridList,
         Frame->OptionValue[Frame],
         Alignment->OptionValue[Alignment],
         Frame->OptionValue[Frame],
         ItemSize->OptionValue[ItemSize]
         ]
   )
];

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.";
Options[HamiltonianForm]={"Separator"->"","Pivot"->""};
HamiltonianForm[hamMatrix_, basisLabels_List, OptionsPattern[]] :=
  (
     braLabels=DisplayForm[RowBox[{"\[LeftAngleBracket]",#,"\[
  RightBracketingBar]"}]]& /@ basisLabels;
     ketLabels=DisplayForm[RowBox[{"\[LeftBracketingBar]",#,"\[
  RightAngleBracket]"}]]& /@ basisLabels;
     LabeledGrid[hamMatrix,braLabels,ketLabels,"Separator"->
  OptionValue["Separator"],"Pivot"->OptionValue["Pivot"]]
     )

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.";
Options[HamiltonianMatrixPlot] = Join[Options[MatrixPlot], {"Hover"
   -> True, "Overlay Values" -> True}];
HamiltonianMatrixPlot[hamMatrix_, basisLabels_, opts :
  OptionsPattern[]] := (
   braLabels = DisplayForm[RowBox[{"\[LeftAngleBracket]", #, "\[
  RightBracketingBar]"}]] & /@ basisLabels;
   ketLabels = DisplayForm[Rotate[RowBox[{"\[LeftBracketingBar]", #,
    "\[RightAngleBracket]"}],\[Pi]/2]] & /@ basisLabels;
   ketLabelsUpright = DisplayForm[RowBox[{"\[LeftBracketingBar]", #,
    "\[RightAngleBracket]"}]] & /@ basisLabels;
   numRows = Length[hamMatrix];
   numCols = Length[hamMatrix[[1]]];
   epiThings = Which[
     And[OptionValue["Hover"], Not[OptionValue["Overlay Values"]]],
     Flatten[
       Table[
         Tooltip[
           {
             Transparent,
             Rectangle[
             {j - 1, numRows - i},
```

```
4380                {j - 1, numRows - i} + {1, 1}
4381                ]
4382              },
4383            Row[{braLabels[[i]],ketLabelsUpright[[j]],"=",hamMatrix[[i,
     j]]}]
4384            ],
4385          {i, 1, numRows},
4386          {j, 1, numCols}
4387          ]
4388        ],
4389        And[OptionValue["Hover"], OptionValue["Overlay Values"]],
4390        Flatten[
4391          Table[
4392            Tooltip[
4393              {
4394                Transparent,
4395                Rectangle[
4396                {j - 1, numRows - i},
4397                {j - 1, numRows - i} + {1, 1}
4398                ]
4399              },
4400            DisplayForm[RowBox[{"\[LeftAngleBracket]", basisLabels[[i
    ]], "\[LeftBracketingBar]", basisLabels[[j]], "\[RightAngleBracket
    ]"}]]
4401            ],
4402          {i, numRows},
4403          {j, numCols}
4404          ]
4405        ],
4406        True,
4407        {}
4408        ];
4409      textOverlay = If[OptionValue["Overlay Values"],
4410        (
4411          Flatten[
4412          Table[
4413            Text[hamMatrix[[i, j]],
4414              {j - 1/2, numRows - i + 1/2}
4415            ],
4416          {i, 1, numRows},
4417          {j, 1, numCols}
4418          ]
4419          ]
4420        ),
4421        {}
4422        ];
4423      epiThings = Join[epiThings, textOverlay];
4424      MatrixPlot[hamMatrix,
4425        FrameTicks -> {
4426          {Transpose[{Range[Length[braLabels]], braLabels}], None},
4427          {None, Transpose[{Range[Length[ketLabels]], ketLabels}]}
4428          },
4429        Evaluate[FilterRules[{opts}, Options[MatrixPlot]]],
4430        Epilog -> epiThings
4431      ]
4432    );
4433
4434    (* ################## Some Plotting Routines ################### *)
4435    (* ############################################################ *)
4436
4437    (* ############################################################ *)
4438    (* ###################### Load Functions ###################### *)
4439
4440    LoadAll::usage = "LoadAll[] executes most Load* functions.";
4441    LoadAll[] := (
4442      LoadTermLabels[];
4443      LoadCFP[];
4444      LoadUk[];
4445      LoadV1k[];
4446      LoadT22[];
4447      LoadSOOandECSOLS[];
4448
4449      LoadElectrostatic[];
4450      LoadSpinOrbit[];
4451      LoadSOOandECSO[];
4452      LoadSpinSpin[];
```

```
4453    LoadThreeBody[];
4454    LoadChenDeltas[];
4455    LoadCarnall[];
4456  );
4457
4458  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.";
4459
4460  LoadTermLabels::usage = "LoadTermLabels[] loads into the session
        the labels for the terms in the f^n configurations.";
4461  LoadTermLabels[] := (
4462    If[ValueQ[fnTermLabels], Return[]];
4463    PrintTemporary["Loading data for state labels in the f^n
      configurations..."];
4464    fnTermsFname = FileNameJoin[{moduleDir, "data", "fnTerms.m"}];
4465
4466    If[!FileExistsQ[fnTermsFname],
4467      (PrintTemporary[">> fnTerms.m not found, generating ..."];
4468        fnTermLabels = ParseTermLabels["Export"->True];
4469      ),
4470      fnTermLabels = Import[fnTermsFname];
4471    ];
4472  );
4473
4474  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}";
4475
4476  LoadCarnall::usage = "LoadCarnall[] loads data for trivalent
        lanthanides in LaF3 using the data from Bill Carnall's 1989 paper.
        ";
4477  LoadCarnall[] := (
4478    If[ValueQ[Carnall], Return[]];
4479    carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
4480    If[!FileExistsQ[carnallFname],
4481      (PrintTemporary[">> Carnall.m not found, generating ..."];
4482        Carnall = ParseCarnall[];
4483      ),
4484      Carnall = Import[carnallFname];
4485    ];
4486  );
4487
4488  LoadChenDeltas::usage = "LoadChenDeltas[] loads the differences
        noted by Chen.";
4489  LoadChenDeltas[] := (
4490    If[ValueQ[chenDeltas], Return[]];
4491    PrintTemporary["Loading the association of discrepancies found by
         Chen ..."];
4492    chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas.m"
      }];
4493    If[!FileExistsQ[chenDeltasFname],
4494      (PrintTemporary[">> chenDeltas.m not found, generating ..."];
4495        chenDeltas = ParseChenDeltas[];
4496      ),
4497      chenDeltas = Import[chenDeltasFname];
4498    ];
4499  );
4500
4501  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";
4502  Options[ParseChenDeltas] = {"Export" -> True};
4503  ParseChenDeltas[OptionsPattern[]] := (
4504    chenDeltasRaw = Import[FileNameJoin[{moduleDir, "data", "the-chen
      -deltas-A.csv"}]];
4505    chenDeltasRaw = chenDeltasRaw[[2 ;;]];
4506    chenDeltas = <||>;
4507    chenDeltasA = <||>;
4508    Off[Power::infy];
4509    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;
        Carnall[keys[[4]]]  = headers;
        ),
    {sheetIdx,4,16}
    ];

    goodions = Select[ions,#!="Pm"&];
    expData  = Select[Transpose[Carnall["appendix:"<>#<>":RawTable"
```

```
    ]][[1+Position[Carnall["appendix:"<>#<>":Headings"],"Exp (1/cm)"
    ][[1,1]]]],NumberQ]&/@goodions;
4639    Carnall["All Experimental Data"]=AssociationThread[goodions,
    expData];
4640    If[OptionValue["Export"],
4641      (
4642        carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"
    }];
4643        Print["Exporting to "<>carnallFname];
4644        Export[carnallFname, Carnall];
4645      )
4646      ];
4647    Return[Carnall];
4648  );
4649
4650  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";
4651
4652  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[].";
4653
4654  LoadCFP::usage = "LoadCFP[] loads CFP, CFPAssoc, and CFPTable into
    the session.";
4655  LoadCFP[] := (
4656    If[And[ValueQ[CFP], ValueQ[CFPTable], ValueQ[CFPAssoc]],Return
    []];
4657
4658    PrintTemporary["Loading CFPtable ..."];
4659    CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"}];
4660    If[!FileExistsQ[CFPTablefname],
4661      (PrintTemporary[">> CFPTable.m not found, generating ..."];
4662        CFPTable = GenerateCFPTable["Export"->True];
4663      ),
4664      CFPTable = Import[CFPTablefname];
4665    ];
4666
4667    PrintTemporary["Loading CFPs.m ..."];
4668    CFPfname = FileNameJoin[{moduleDir, "data", "CFPs.m"}];
4669    If[!FileExistsQ[CFPfname],
4670      (PrintTemporary[">> CFPs.m not found, generating ..."];
4671        CFP = GenerateCFP["Export"->True];
4672      ),
4673      CFP = Import[CFPfname];
4674    ];
4675
4676    PrintTemporary["Loading CFPAssoc.m ..."];
4677    CFPAfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"}];
4678    If[!FileExistsQ[CFPAfname],
4679      (PrintTemporary[">> CFPAssoc.m not found, generating ..."];
4680        CFPAssoc = GenerateCFPAssoc["Export"->True];
4681      ),
4682      CFPAssoc = Import[CFPAfname];
4683    ];
4684  );
4685
4686  ReducedUkTable::usage = "ReducedUkTable[{n, l = 3, SL, SpLp, k}]
    provides reduced matrix elements of the unit spherical tensor
    operator Uk. See TASS section 11-9 \"Unit Tensor Operators\".
    Loaded using LoadUk[].";
4687
4688  LoadUk::usage = "LoadUk[] loads into session the reduced matrix
    elements for unit tensor operators.";
4689  LoadUk[] := (
4690    If[ValueQ[ReducedUkTable], Return[]];
4691    PrintTemporary["Loading the association of reduced matrix
    elements for unit tensor operators ..."];
4692    ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
    ReducedUkTable.m"}];
4693    If[!FileExistsQ[ReducedUkTableFname],
```

```
4694        (PrintTemporary[">> ReducedUkTable.m not found, generating ..."
      ];
4695          ReducedUkTable = GenerateReducedUkTable[7];
4696        ),
4697        ReducedUkTable = Import[ReducedUkTableFname];
4698      ];
4699    );
4700
4701    ElectrostaticTable::usage = "ElectrostaticTable[{n, SL, SpLp}]
      provides the calculated result of Electrostatic[{n, SL, SpLp}].
      Load using LoadElectrostatic[].";
4702
4703    LoadElectrostatic::usage = "LoadElectrostatic[] loads the reduced
      matrix elements for the electrostatic interaction.";
4704    LoadElectrostatic[] := (
4705      If[ValueQ[ElectrostaticTable], Return[]];
4706      PrintTemporary["Loading the association of matrix elements for
      the electrostatic interaction ..."];
4707      ElectrostaticTablefname = FileNameJoin[{moduleDir, "data", "
      ElectrostaticTable.m"}];
4708      If[!FileExistsQ[ElectrostaticTablefname],
4709        (PrintTemporary[">> ElectrostaticTable.m not found, generating
      ..."];
4710          ElectrostaticTable = GenerateElectrostaticTable[7];
4711        ),
4712        ElectrostaticTable = Import[ElectrostaticTablefname];
4713      ];
4714    );
4715
4716    LoadV1k::usage = "LoadV1k[] loads into session the matrix elements
      of V1k.";
4717    LoadV1k[] := (
4718      If[ValueQ[ReducedV1kTable], Return[]];
4719      PrintTemporary["Loading the association of matrix elements for
      V1k ..."];
4720      ReducedV1kTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedV1kTable.m"}];
4721      If[!FileExistsQ[ReducedV1kTableFname],
4722        (PrintTemporary[">> ReducedV1kTable.m not found, generating ...
      "];
4723          ReducedV1kTable = GenerateReducedV1kTable[7];
4724        ),
4725        ReducedV1kTable = Import[ReducedV1kTableFname];
4726      ]
4727    );
4728
4729    LoadSpinOrbit::usage = "LoadSpinOrbit[] loads into session the
      matrix elements of the spin-orbit interaction.";
4730    LoadSpinOrbit[] := (
4731      If[ValueQ[SpinOrbitTable], Return[]];
4732      PrintTemporary["Loading the association of matrix elements for
      spin-orbit ..."];
4733      SpinOrbitTableFname = FileNameJoin[{moduleDir, "data", "
      SpinOrbitTable.m"}];
4734      If[!FileExistsQ[SpinOrbitTableFname],
4735        (PrintTemporary[">> SpinOrbitTable.m not found, generating ..."
      ];
4736          SpinOrbitTable = GenerateSpinOrbitTable[7, "Export" -> True];
4737        ),
4738        SpinOrbitTable = Import[SpinOrbitTableFname];
4739      ]
4740    );
4741
4742    LoadSOOandECSOLS::usage = "LoadSOOandECSOLS[] loads into session
      the LS reduced matrix elements of the SOO-ECSO interaction.";
4743    LoadSOOandECSOLS[] := (
4744      If[ValueQ[SOOandECSOLSTable], Return[]];
4745      PrintTemporary["Loading the association of LS reduced matrix
      elements for SOO-ECSO ..."];
4746      SOOandECSOLSTableFname = FileNameJoin[{moduleDir, "data", "
      ReducedSOOandECSOLSTable.m"}];
4747      If[!FileExistsQ[SOOandECSOLSTableFname],
4748        (PrintTemporary[">> ReducedSOOandECSOLSTable.m not found,
      generating ..."];
4749          SOOandECSOLSTable = GenerateSOOandECSOLSTable[7];
4750        ),
```

```
4751        SOOandECSOLSTable = Import[SOOandECSOLSTableFname];
4752      ];
4753    );
4754
4755    LoadSOOandECSO::usage = "LoadSOOandECSO[] loads into session the
          LSJ reduced matrix elements of spin-other-orbit and
          electrostatically-correlated-spin-orbit.";
4756    LoadSOOandECSO[] := (
4757      If[ValueQ[SOOandECSOTableFname], Return[]];
4758      PrintTemporary["Loading the association of matrix elements for
          spin-other-orbit and electrostatically-correlated-spin-orbit ..."
          ];
4759      SOOandECSOTableFname = FileNameJoin[{moduleDir, "data", "
          SOOandECSOTable.m"}];
4760      If[!FileExistsQ[SOOandECSOTableFname],
4761        (PrintTemporary[">> SOOandECSOTable.m not found, generating ...
          "];
4762          SOOandECSOTable = GenerateSOOandECSOTable[7, "Export"->True];
4763        ),
4764        SOOandECSOTable = Import[SOOandECSOTableFname];
4765      ];
4766    );
4767
4768    LoadT22::usage = "LoadT22[] loads into session the matrix elements
          of T22.";
4769    LoadT22[] := (
4770      If[ValueQ[T22Table], Return[]];
4771      PrintTemporary["Loading the association of reduced T22 matrix
          elements ..."];
4772      T22TableFname = FileNameJoin[{moduleDir, "data", "ReducedT22Table
          .m"}];
4773      If[!FileExistsQ[T22TableFname],
4774        (PrintTemporary[">> ReducedT22Table.m not found, generating ...
          "];
4775          T22Table = GenerateT22Table[7];
4776        ),
4777        T22Table = Import[T22TableFname];
4778      ];
4779    );
4780
4781    LoadSpinSpin::usage = "LoadSpinSpin[] loads into session the matrix
          elements of spin-spin.";
4782    LoadSpinSpin[] := (
4783      If[ValueQ[SpinSpinTable], Return[]];
4784      PrintTemporary["Loading the association of matrix elements for
          spin-spin ..."];
4785      SpinSpinTableFname = FileNameJoin[{moduleDir, "data", "
          SpinSpinTable.m"}];
4786      If[!FileExistsQ[SpinSpinTableFname],
4787        (PrintTemporary[">> SpinSpinTable.m not found, generating ..."
          ];
4788          SpinSpinTable = GenerateSpinSpinTable[7, "Export" -> True];
4789        ),
4790        SpinSpinTable = Import[SpinSpinTableFname];
4791      ];
4792    );
4793
4794    LoadThreeBody::usage = "LoadThreeBody[] loads into session the
          matrix elements of three-body configuration-interaction effects.";
4795    LoadThreeBody[] := (
4796      If[ValueQ[ThreeBodyTable], Return[]];
4797      PrintTemporary["Loading the association of matrix elements for
          three-body configuration-interaction effects ..."];
4798      ThreeBodyFname   = FileNameJoin[{moduleDir, "data", "
          ThreeBodyTable.m"}];
4799      ThreeBodiesFname = FileNameJoin[{moduleDir, "data", "
          ThreeBodyTables.m"}];
4800      If[!FileExistsQ[ThreeBodyFname],
4801        (PrintTemporary[">> ThreeBodyTable.m not found, generating ..."
          ];
4802          {ThreeBodyTable, ThreeBodyTables} = GenerateThreeBodyTables
          [14, "Export" -> True];
4803        ),
4804        ThreeBodyTable = Import[ThreeBodyFname];
4805        ThreeBodyTables = Import[ThreeBodiesFname];
4806      ];
```

```
  );

  (* #################### 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 *)
    coarseStates  = {#[[1]],#[[-1]]}& /@ coarseStates;
    missingLevels = Length[coarseStates]-Length[expData];

    (* The energies are in the first element of the tuples. *)
    energyDiffFun = ( Abs[#1[[1]]-#2[[1]]] ) &;

    (* matching disregarding labels to get overall scale for scaling
    differences in labels *)
    energyFlow    = FlowMatching[coarseStates,
                      expData,
                      \"notMatched\" -> missingLevels,
                      \"CostFun\"    -> energyDiffFun
                      ];
    energyPairs   = {#[[1]][[1]], #[[2]][[1]]}&/@energyFlow[[1]];
    energyRms     = Sqrt[Total[(Abs[#[[2]]-#[[1]]])^2 & /@
    energyPairs]/Length[energyPairs]];

    (* matching using both labels and energies  *)
    energyAndLabelFun  = With[{del=energyRms},
                (Abs[#1[[1]]-#2[[1]]] +
                If[#1[[2]]==#2[[2]],
                  0.,
                  del])&];

    (* energyAndLabelFun  = With[{del=energyRms},
                (Abs[#1[[1]]-#2[[1]]] +
                del*EditDistance[#1[[2]],#2[[2]]])&]; *)
    energyAndLabelFun  = ( Abs[#1[[1]] - #2[[1]]] + EditDistance
    [#1[[2]],#2[[2]]] )&;
    energyAndLabelFlow = FlowMatching[coarseStates,
                         expData,
                         \"notMatched\" -> missingLevels,
                         \"CostFun\"    -> energyAndLabelFun
                         ];
    totalAvgCost       = Total[energyAndLabelFun@@# & /@
    energyAndLabelFlow[[1]]]/Length[energyAndLabelFlow[[1]]];
    Return[totalAvgCost];
  )
]"];

truncatedEnergyCostTemplate = StringTemplate["
TheTruncatedAndSignedPath[`varsWithNumericQ`] :=
```

```
(
  (* Calculate the truncated Hamiltonian *)
  numericalFreeIonHam = compileIntermediateTruncatedHam[`
    varsMixedWithFixedVals`];

  (* Diagonalize it *)
  {truncatedEigenvalues, truncatedEigenVectors} = Eigensystem[
    numericalFreeIonHam];

  (* Using the truncated eigenvectors push them up to the full state
    space *)
  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
```

```
346 The PrintFun option is the function used to print progress messages.
347 The FilePrefix option is the prefix to use for the file name, by
    default this is the symbol for the lanthanide.
348 The SlackChannel option is the channel to post progress messages to.
349 The MaxHistory option is the maximum number of iterations to keep in
    the history.
350 The MaxIters option is the maximum number of iterations for the
    solver.
351 The NumCycles option is the number of cycles to run the solver for.
352 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
353 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.";
354 Options[FitToHam] = {
355       "PrintFun"-> PrintTemporary,
356       "FilePrefix" -> "",
357       "SlackChannel" -> None,
358       "MaxHistory" -> 100,
359       "ProgressWindow"-> True,
360       "MaxIters" -> 100,
361       "NumCycles" -> 10};
362 FitToHam[numE_Integer, expData_List, fitToSymbols_List,
    simplifier_List, OptionsPattern[]] :=
363   (
364     PrintFun         = OptionValue["PrintFun"];
365     fitToVars        = ToExpression[ToString[#] <> "v"] & /@
    fitToSymbols;
366     stringfitToVars  = ToString /@ fitToVars;
367     slackChan        = OptionValue["SlackChannel"];
368     maxHistory       = OptionValue["MaxHistory"];
369     maxIters         = OptionValue["MaxIters"];
370     numCycles        = OptionValue["NumCycles"];
371     ln               = theLanthanides[[numE]];
372     logFilePrefix    = If[OptionValue["FilePrefix"] == "",
373                           ToString[theLanthanides[[numE]]],
374                           OptionValue["FilePrefix"]];
375     PrintFun["Assembling the Hamiltonian for f^", numE, " ..."];
376     ham = HamMatrixAssembly[numE];
377     PrintFun["Simplifying the symbolic expression for the Hamiltonian
     in terms of the given simplifier..."];
378     ham = ReplaceInSparseArray[ham, simplifier];
379     PrintFun["Determining the variables to be fit for ..."];
380     (* as they remain after simplifying *)
381     fitVars = Variables[Normal[ham]];
382     (* append v to symbols *)
383     varVars = ToExpression[ToString[#] <> "v"] & /@ fitVars;
384
385     PrintFun[
386     "Compiling a function for efficient evaluation of the Hamiltonian
     matrix ..."];
387     compHam = Compile[Evaluate[fitVars], Evaluate[N[Normal[ham]]]];
388
389     PrintFun[
390     "Defining the cost function according to given energies and state
     labels ..."];
391
392     varPatterns = StringJoin[{ToString[#], "_?NumericQ"}] & /@
    fitVars;
393     varPatterns = Riffle[varPatterns, ", "];
394     varPatterns = StringJoin[varPatterns];
395     vars = ToString[#] & /@ fitVars;
396     vars = Riffle[vars, ", "];
397     vars = StringJoin[vars];
398
399     basis = BasisLSJMJ[numE];
400
401     (* define the cost functions given the problem variables *)
402     energyCostFunString =
403     energyCostFunTemplate[<|
404       "varPatterns" -> varPatterns,
405       "vars" -> vars|>];
406     ToExpression[energyCostFunString];
407     energyAndLabelCostFunString = energyAndLabelCostFunTemplate[<|"
```

```
    varPatterns" -> varPatterns, "vars" -> vars|>];
408 ToExpression[energyAndLabelCostFunString];
409
410 PrintFun["getting starting values from LaF3..."];
411 lnParams = LoadParameters[ln];
412 bills = Table[lnParams[ToExpression[StringTake[ToString[varvar],
    {1, -2}]]], {varvar, varVars}];
413
414 (* define the function arguments with the frozen args in place *)
415 activeArgs = Table[
416   If[MemberQ[fitToVars, varvar],
417   varvar,
418   lnParams[ToExpression[StringTake[ToString[varvar], {1, -2}]]]],
    {varvar, varVars}
420 ];
421 activeArgs = StringJoin[Riffle[ToString /@ activeArgs, ", "]];
422 (* the constraints, very important *)
423 constraints = N[Constrainer[fitToVars, ln]];
424 complementaryArgs = Table[
425   If[MemberQ[fitToVars, varvar],
426   varvar,
427   lnParams[ToExpression[StringTake[ToString[varvar], {1, -2}]]]],
    {varvar, varVars}
429 ];
430
431 fromBill = {B02v -> B02, B04v -> B04, B06v -> B06, B22v -> B22,
    B24v -> B24, B26v -> B26, B44v -> B44, B46v -> B46, B66v -> B66,
    M0v -> M0, P2v -> P2} /. lnParams;
434
435 If[Not[ValueQ[noteboo]] && OptionValue["ProgressWindow"],
    noteboo = ProgressNotebook["Basic" -> False];
437 ];
438
439 threadHeaderTemplate = StringTemplate[
    "(‘idx‘/‘reps‘) Fitting data for ‘ln‘ using ‘freeVars‘."
441 ];
442 solutions = {};
443 Do[
444 (
445   (* Remove the downvalues of the cost function *)
446   (* DownValues[TheRightSignedPath] = {DownValues[
    TheRightSignedPath][[-1]]}; *)
447   (* start history anew *)
448   rmsHistory = {};
449   paramSols  = {};
450   startTime  = Now;
451   threadMessage = threadHeaderTemplate[
452       <|"reps" -> numCycles,
453       "idx" -> rep,
454       "ln" -> ln,
455       "freeVars" -> ToString[fitToVars]|>];
456   If[slackChan =!= None,
457     threadTS = PostMessageToSlack[threadMessage, slackChan]["ts"]
458   ];
459   solverTemplateNMini = StringTemplate["
460     numIter = 0;
461     sol = NMinimize[
462       Evaluate[
463         Join[{TheRightSignedPath[‘activeArgs‘]},
464           constraints
465         ]
466       ],
467       fitToVars,
468       MaxIterations->‘maxIterations‘,
469       Method->‘Method‘,
470       ‘Monitor‘:>(
471         currentErr = TheRightSignedPath[‘activeArgs‘];
472         numIter   += 1;
473         rmsHistory = AddToList[rmsHistory, currentErr, maxHistory
    , False];
474         paramSols  = AddToList[paramSols, fitToVars, maxHistory,
    False];
475       )
476     ]
477   "];
478   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;
      atable = TableForm[compa,
        TableHeadings -> {None,
          {"ql", "ql", "Bill (exp)", "Bill (exp)",
          "Bill (calc)", "labels=", "ql - exp", "bill - exp"}}
      ];
      atable = Framed[atable, FrameMargins -> 20];
      upsAndDowns = {
        {"Better", Length[Select[compa, #[[-1]] == "Better" &]]},
        {"Equal",  Length[Select[compa, #[[-1]] == "Equal" &]]},
        {"Worse",  Length[Select[compa, #[[-1]] == "Worse" &]]}
      };
      upsAndDowns = TableForm[upsAndDowns];
      If[slackChan =!= None,
        PostPdfToSlack["table", atable, slackChan, "threadTS" ->
    threadTS];
      ];
      solutions = Append[solutions, sol];
    ),
    {rep, 1, numCycles}
    ];
  )

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\".

The following options can be used:
  \"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.
  \"FilePrefix\" : the prefix to use for the file name, by default
    this is the symbol for the lanthanide.
  \"AccuracyGoal\": sets the accuracy goal for NMinimize, the default
     is 3.
  \"MaxHistory\" : determines how long the logs for the solver can be
    .
  \"MaxIterations\": determines the maximum number of iterations used
    by NMinimize.

```

```
592   \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
        3.
593   \"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.
594   \"PrintFun\": the function used to print progress messages, the
        default is PrintTemporary.
595   \"SlackChannel\": name of the Slack channel to which to dump
        progress messaages, the default is None which disables this option
         entirely.
596   \"ProgressView\": whether or not a progress window will be opened
        to show the progress of the solver, the default is True.
597
598   \"ReturnHashFileNameAndExit\": if True then the function returns
        the name of the file with the solutions and exits, the default is
        False.
599   \"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 {}.
600   \"DE:CrossProbability\": the cross probability used by the
        Differential Evolution method, the default is 0.5.
601   \"DE:ScalingFactor\": the scaling factor used by the Differential
        Evolution method, the default is 0.6.
602   \"DE:SearchPoints\": the number of search points used by the
        Differential Evolution method, the default is Automatic.
603
604   \"MagneticSimplifier\": a list of replacement rules to simplify the
         Marvin and pesudo-magnetic paramters.
605   \"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.
606   \"SymmetrySimplifier\": a list of replacements rules to simplify
        the crystal field.
607   \"OtherSimplifier\": an additiona list of replacement rules that
        are applied to the Hamiltonian before computing with it.
608   \"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.
609
610   \"FreeIonSymbols\": a list with the symbols to be included in the
        intermediate coupling basis.
611   \"AppendToLogFile\": an association appended to the log file under
        the key \"Appendix\".
612   ";
613 Options[TruncationFit]={
614   "MaxHistory"       -> 200,
615   "MaxIterations"    -> 100,
616   "FilePrefix"       -> "",
617   "AccuracyGoal"     -> 3,
618   "TruncationEnergy" -> Automatic,
619   "PrintFun"         -> PrintTemporary,
620   "SlackChannel"     -> None,
621   "ProgressView"     -> True,
622   "SignatureCheck"   -> False,
623   "AppendToLogFile"  -> <||>,
624   "StartingForVars"  -> {},
625   "ReturnHashFileNameAndExit" -> False,
626   "DE:CrossProbability" -> 0.5,
627   "DE:ScalingFactor"    -> 0.6,
628   "DE:SearchPoints"     -> Automatic,
629   "MagneticSimplifier"  -> {
630     M2 -> 56/100 M0,
631     M4 -> 31/100 M0,
632     P4 -> 1/2 P2,
633     P6 -> 1/10 P2},
634   "MagFieldSimplifier" -> {
635     Bx->0,By->0,Bz->0
636     },
637   "SymmetrySimplifier" -> {
638     B12->0,B14->0,B16->0,B34->0,B36->0,B56->0,
639     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, M0, P2, α, β, γ, ζ, T2, T3, T4,
    T6, T7, T8}
};
TruncationFit[numE_Integer, expData0_List, numReps_Integer,
    activeVars_List, startingValues_List, OptionsPattern[]]:=(
  ln               = theLanthanides[[numE]];
  expData          = expData0;
  PrintFun         = OptionValue["PrintFun"];
  truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic,
    Max[First/@expData],
    OptionValue["TruncationEnergy"]
  ];
  oddsAndEnds    = <||>;
  expData        = Select[expData, #[[1]] <= truncationEnergy &];
  maxIterations  = OptionValue["MaxIterations"];
  maxHistory     = OptionValue["MaxHistory"];
  slackChan      = OptionValue["SlackChannel"];
  accuracyGoal   = OptionValue["AccuracyGoal"];
  logFilePrefix  = If[OptionValue["FilePrefix"] == "",
                      ToString[theLanthanides[[numE]]],
                      OptionValue["FilePrefix"]];

  usingInitialRange  = Not[OptionValue["StartingForVars"] === {}];
  If[usingInitialRange,
    (
      PrintFun["Using the solver for initial values in range ..."];
      {fractionalWidth, startVarValues} = OptionValue["
    StartingForVars"];
    )
  ];

  magneticSimplifier  = OptionValue["MagneticSimplifier"];
  magFieldSimplifier  = OptionValue["MagFieldSimplifier"];
  symmetrySimplifier  = OptionValue["SymmetrySimplifier"];
  otherSimplifier     = OptionValue["OtherSimplifier"];
  threeBodySimplifier = If[Head[OptionValue["ThreeBodySimplifier"]]
    == Association,
    OptionValue["ThreeBodySimplifier"][numE],
    OptionValue["ThreeBodySimplifier"]
  ];
  simplifier          = Join[magneticSimplifier,
                        magFieldSimplifier,
```

```
                        symmetrySimplifier,
                        threeBodySimplifier,
                        otherSimplifier];
freeIonSymbols = OptionValue["FreeIonSymbols"];
runningInteractive = (Head[$ParentLink] === LinkObject);

oddsAndEnds["simplifier"] = simplifier;
oddsAndEnds["freeIonSymbols"] = freeIonSymbols;
oddsAndEnds["truncationEnergy"] = truncationEnergy;
oddsAndEnds["numE"] = numE;
oddsAndEnds["expData"] = expData;
oddsAndEnds["numReps"] = numReps;
oddsAndEnds["activeVars"] = activeVars;
oddsAndEnds["startingValues"] = startingValues;
oddsAndEnds["maxIterations"] = maxIterations;
oddsAndEnds["PrintFun"] = PrintFun;
oddsAndEnds["ln"] = ln;
oddsAndEnds["numE"] = numE;
oddsAndEnds["accuracyGoal"] = accuracyGoal;
oddsAndEnds["Appendix"] = OptionValue["AppendToLogFile"];

hamDim         = Binomial[14, numE];
modelSymbols   = Sort[Select[paramSymbols, Not[MemberQ[Join[
  racahSymbols, chenSymbols,{t2Switch, \[Epsilon],gs}],#]]&]];
(* Remove the symbols that will be removed by the simplifier, no
  symbol should remain here that is not in the symbolic hamiltonian
  *)
reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
  simplifier],#]]&];
If[OptionValue["SignatureCheck"],
  (
  Print["Given the model parameters and the simplifying assumptions
  , the resultant model parameters are:"];
  Print[{reducedModelSymbols}];
  Print["The ordering in these needs to be respected in the
  startValues parameter ..."];
  Print["Exiting ..."];
  Return[""];
  )
];

(*calculate the basis*)
basis          = BasisLSJMJ[numE];
(* grab the Hamiltonian preserving its block structure *)
PrintFun["Assembling the Hamiltonian for f^",numE," keeping the
  block structure ..."];
ham            = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
(* apply the simplifier *)
PrintFun["Simplifying using the given aggregate set of
  simplification rules ..."];
ham            = Map[ReplaceInSparseArray[#, simplifier]&, ham, {2}];

(* Get the reference parameters from LaF3 *)
PrintFun["Getting reference parameters for ",ln," using LaF3 ..."];
lnParams = LoadParameters[ln];
freeBies = Prepend[Values[(#->(#/.lnParams))&/@freeIonSymbols],numE
  ];
(* a more explicit alias *)
allVars            = reducedModelSymbols;

oddsAndEnds["allVars"]  = allVars;
oddsAndEnds["freeBies"] = freeBies;

(* reload compiled version if found *)
varHash                   = Hash[{numE, allVars, freeBies,
  truncationEnergy}];
compileIntermediateFname = "compileIntermediateTruncatedHam-"<>
  ToString[varHash]<>".mx";
truncadedFname            = "TheTruncatedAndSignedPath-"<>ToString[
  varHash]<>".mx";
If[OptionValue["ReturnHashFileNameAndExit"],
  (
    Print[varHash];
    Return[truncadedFname];
  )
];
```

```
771  If[FileExistsQ[compileIntermediateFname],
772    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 ..."];
773    {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
     Import[compileIntermediateFname];,
774  (
775    PrintFun["Zeroing out every symbol in the Hamiltonian that is not
     a free-ion parameter ..."];
776    (* Get the free ion symbols *)
777    freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
     freeIonSymbols];
778    (* Take the diagonal blocks for the intermediate analysis *)
779    PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."];
780    diagonalBlocks       = Diagonal[ham];
781    (* simplify them to only keep the free ion symbols *)
782    PrintFun["Simplifying the diagonal blocks to only keep the free
     ion symbols ..."];
783    diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
     ]&/@diagonalBlocks;
784    (* these include the MJ quantum numbers, remove that *)
785    PrintFun["Contracting the basis vectors by removing the MJ
     quantum numbers from the diagonal blocks ..."];
786    diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
787
788    argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
     Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
789    argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
     ToString[#]<>"v") & /@ freeIonSymbols,", "]];
790    PrintFun["argsOfTheIntermediateEigensystems = ",
     argsOfTheIntermediateEigensystems];
791    PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
     argsForEvalInsideOfTheIntermediateSystems];
792    PrintFun["If the following fails, make sure to modify the
     arguments of TheIntermediateEigensystems to match the ones above
     ..."];
793
794    (* 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 *)
795    (* Compile one function for each of the blocks *)
796    PrintFun["Compiling functions for the diagonal blocks of the
     Hamiltonian ..."];
797    compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N[
     Normal[#]]]]&/@diagonalScalarBlocks;
798    (* Use that to create a function that will calculate the free-ion
      eigensystem *)
799    TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, M0v_,
      P2v_, αv_, βv_, γv_, ζv_, T2v_, T3v_, T4v_, T6v_, T7v_, T8v_] :=
     (
800      theNumericBlocks = (#[F0v, F2v, F4v, F6v, M0v, P2v, αv, βv, γv,
      ζv, T2v, T3v, T4v, T6v, T7v, T8v]&)/@compiledDiagonal;
801      theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
802      Js      = AllowedJ[numEv];
803      basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
804      (* Having calculated the eigensystems with the removed
     degeneracies, put the degeneracies back in explicitly *)
805      elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2Js
     [[#2[[1]]]]+1]&, theIntermediateEigensystems];
806      pivot = If[EvenQ[numEv],0,-1/2];
807      LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/@Select[
     BasisLSJMJ[numEv],#[[-1]]== pivot &];
808      (* Calculate the multiplet assignments that the intermediate
     basis eigenvectors have *)
809      multipletAssignments = Table[
810        (
811          J          = Js[[idx]];
812          eigenVecs = theIntermediateEigensystems[[idx]][[2]];
813          majorComponentIndices        = Ordering[Abs[#]][[-1]]&/
     @eigenVecs;
814          majorComponentAssignments    = LSJmultiplets[[#]]&/
     @majorComponentIndices;
815          (* All of the degenerate eigenvectors belong to the same
     multiplet*)
816          elevatedMultipletAssignments = ListRepeater[
```

```
    majorComponentAssignments,2J+1];
817           elevatedMultipletAssignments
818         ),
819       {idx, 1, Length[Js]}
820       ];
821       (* Put together the multiplet assignments and the energies *)
822       freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
    @elevatedIntermediateEigensystems, multipletAssignments}];
823       freeIenergiesAndMultiplets = Flatten[freeIenergiesAndMultiplets
    , 1];
824       (* Calculate the change of basis matrix using the intermediate
    coupling eigenvectors *)
825       basisChanger = BlockDiagonalMatrix[Transpose/@Last/
    @elevatedIntermediateEigensystems];
826       basisChanger = SparseArray[basisChanger];
827       Return[{theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger}]
828     );
829
830     PrintFun["Calculating the intermediate eigensystems for ",ln,"
    using free-ion params from LaF3 ..."];
831     (* Calculate intermediate coupling basis using the free-ion
    params for LaF3 *)
832     {theIntermediateEigensystems, multipletAssignments,
    elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
    basisChanger} = TheIntermediateEigensystems@@freeBies;
833
834     (* Use that intermediate coupling basis to compile a function for
     the full Hamiltonian *)
835     allFreeEnergies = Flatten[First/@elevatedIntermediateEigensystems
    ];
836     (* Important that the intermediate coupling basis have attached
    energies, which make possible the truncation *)
837     ordering = Ordering[allFreeEnergies];
838     (* Sort the free ion energies and determine which indices should
    be included in the truncation *)
839     allFreeEnergiesSorted          = Sort[allFreeEnergies];
840     {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
841     (* Determine the index at which the energy is equal or larger
    than the truncation energy *)
842     sortedTruncationIndex = Which[
843       truncationEnergy > (maxFreeEnergy-minFreeEnergy),
844       hamDim,
845       True,
846       FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted],
    x_/;x>truncationEnergy,{0},1][[1]]
847     ];
848     (* The actual energy at which the truncation is made *)
849     roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
850
851     (* The indices that enact the truncation *)
852     truncationIndices = ordering[[;;sortedTruncationIndex]];
853
854     (* Using the ham (with all the symbols) change the basis to the
    computed one *)
855     PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
856     intermediateHam            = Transpose[basisChanger].ArrayFlatten
    [ham].basisChanger;
857     (* Using the truncation indices truncate that one *)
858     PrintFun["Truncating the Hamiltonian ..."];
859     truncatedIntermediateHam   = intermediateHam[[truncationIndices,
    truncationIndices]];
860     (* These are the basis vectors for the truncated hamiltonian *)
861     PrintFun["Saving the truncated intermediate basis ..."];
862     truncatedIntermediateBasis = basisChanger[[All,truncationIndices
    ]];
863
864     PrintFun["Compiling a function for the truncated Hamiltonian ..."
    ];
865     (* Compile a function that will calculate the truncated
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
866     compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
```

```
867                                         Evaluate[N[Normal[
      truncatedIntermediateHam]]]];
868     (* Save the compiled function *)
869     PrintFun["Saving the compiled function for the truncated
      Hamiltonian and the truncatedIntermediateBasis..."];
870     Export[compileIntermediateFname, {compileIntermediateTruncatedHam
      , truncatedIntermediateBasis}];
871   )
872   ];
873
874   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.";
875   TheTruncatedAndSignedPathGenerator[varsMixedWithVals_List]:=(
876     variableVars  = Select[varsMixedWithVals,Not[NumericQ[#]]&];
877     numQSignature = StringJoin[Riffle[(ToString[#]<>"_?NumericQ")&/
      @variableVars,", "]];
878     varWithValsSignature = StringJoin[Riffle[(ToString[#]<>"")&/
      @varsMixedWithVals,", "]];
879     funcString    = truncatedEnergyCostTemplate[<|"varsWithNumericQ"
      ->numQSignature,"varsMixedWithFixedVals" -> varWithValsSignature
      |>];
880     ClearAll[TheTruncatedAndSignedPath];
881     ToExpression[funcString]
882   );
883
884   (* We need to create a function call that has all the frozen
      parameters in place and all the active symbols unevaluated *)
885   (* find the indices of the activeVars to create the function
      signature *)
886   activeVarIndices = Flatten[Position[allVars,#]&/@activeVars];
887   (* we start from the numerical values in the current best*)
888   jobVars          = startingValues;
889   (* we then put back the symbols that should be unevaluated *)
890   jobVars[[activeVarIndices]] = activeVars;
891
892   oddsAndEnds["jobVars"] = jobVars;
893   (* calculate the constraints *)
894   constraints               = N[Constrainer[activeVars,ln]];
895   oddsAndEnds["constraints"] = constraints;
896   (* This is useful for the progress window *)
897   activeVarsString  = StringJoin[Riffle[ToString/@activeVars,", "]];
898   TheTruncatedAndSignedPathGenerator[jobVars];
899   stringPartialVars = ToString/@activeVars;
900
901   activeVarsWithRange = If[usingInitialRange,
902     MapIndexed[Flatten[{#1,
903         (1-Sign[startVarValues[[#2]]]*fractionalWidth) *
      startVarValues[[#2]],
904         (1+Sign[startVarValues[[#2]]]*fractionalWidth) *
      startVarValues[[#2]]
905         }]&, activeVars],
906     activeVars
907   ];
908
909   (* this is the template for the minimizer *)
910   solverTemplateNMini = StringTemplate["
911   numIter = 0;
912   sol = NMinimize[
913     Evaluate[
914       Join[{TheTruncatedAndSignedPath[`activeVarsString`]},
915         constraints
916       ]
917     ],
918     activeVarsWithRange,
919     AccuracyGoal    -> `accuracyGoal`,
920     MaxIterations   -> `maxIterations`,
921     Method->`Method`,
922     `Monitor`:>(
923       currentErr = TheTruncatedAndSignedPath[`activeVarsString`];
```

```
      currentParams = activeVars;
      numIter   += 1;
      rmsHistory = AddToList[rmsHistory, currentErr, maxHistory, False];
      paramSols  = AddToList[paramSols, activeVars, maxHistory, False];
      If[Not[runningInteractive],(
        Print[numIter,\"/\",`maxIterations`];
        Print[\"err    = \", ToString[NumberForm[Round[currentErr,0.001],{Infinity,3}]]];
        Print[\"params = \", ToString[NumberForm[Round[#,0.0001],{Infinity,4}]] &/@ currentParams];
        )
      ];
    )
  ]
  "];
  methodStringTemplate = StringTemplate["
      {\"DifferentialEvolution\",
        \"PostProcess\"   -> False,
        \"ScalingFactor\" -> `DE:ScalingFactor`,
        \"CrossProbability\" -> `DE:CrossProbability`,
        \"RandomSeed\"    -> RandomInteger[{0,1000000}],
        \"SearchPoints\"  -> `DE:SearchPoints`}"];
  methodString = methodStringTemplate[<|
    "DE:ScalingFactor" -> OptionValue["DE:ScalingFactor"],
    "DE:CrossProbability" -> OptionValue["DE:CrossProbability"],
    "DE:SearchPoints" -> OptionValue["DE:SearchPoints"]|>];
  (* Evaluate the template *)
  solverCode = solverTemplateNMini[<|
    "accuracyGoal"  -> accuracyGoal,
    "maxIterations" -> maxIterations,
    "Method"->"{\"DifferentialEvolution\",
                \"PostProcess\"   -> False,
                \"ScalingFactor\" -> 0.6,
                \"CrossProbability\" -> 0.25,
                \"RandomSeed\"    -> RandomInteger[{0,1000000}],
                \"SearchPoints\"  -> Automatic}",
    "Monitor"->"StepMonitor",
    "activeVarsString"->activeVarsString|>
  ];
  threadHeaderTemplate = StringTemplate[ "(`idx`/`reps`) Fitting data for `ln` using `freeVars`."];
  (* Find as many solutions as numReps *)
  sols = Table[(
    rmsHistory        = {};
    paramSols         = {};
    openNotebooks     = If[runningInteractive,
              ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks[],
              {}];
    If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["ProgressView"],
      ProgressNotebook["Basic"->False]
    ];
    If[Not[slackChan === None],
    (
      threadMessage = threadHeaderTemplate[<|"reps" -> numReps, "idx" -> rep, "ln" -> ln,
        "freeVars" -> ToString[activeVars]|>];
      threadTS = PostMessageToSlack[threadMessage, slackChan]["ts"];
    )
    ];
    startTime = Now;
    ToExpression[solverCode];

    timeTaken = QuantityMagnitude[Now - startTime, "Seconds"];
    Print["Took " <> ToString[timeTaken] <> "s"];
    Print[sol];
    bestError     = sol[[1]];
    bestParams    = sol[[2]];
    resultMessage = "sigma=" <> ToString[Round[bestError, 0.1]];
    solAssoc = <|
      "bestRMS"      -> bestError,
      "solHistory"   -> rmsHistory,
      "bestParams"   -> bestParams,
```

```
      "paramHistory" -> paramSols,
      "timeTaken/s"    -> timeTaken
      |>;
    solAssoc = Join[solAssoc,   oddsAndEnds];
    logFname = LogSol[solAssoc, logFilePrefix];

    If[Not[slackChan===None],(
      PostMessageToSlack[resultMessage,   slackChan, "threadTS" ->
    threadTS];
      PostFileToSlack[StringSplit[logFname,"/"][[-1]], logFname,
    slackChan, "threadTS" -> threadTS]
    )
    ];
    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.
1025
1026 The function returns an association with the following keys:
1027
1028 \"bestRMS\" which is the best \[Sigma] value found.
1029 \"bestParams\" which is the best set of parameters found.
1030 \"paramSols\" which is a list of the parameters during the stepping
    of the fitting algorithm.
1031 \"timeTaken/s\" which is the time taken to find the best fit.
1032 \"simplifier\" which is the simplifier used to simplify the
    Hamiltonian.
1033 \"excludeDataIndices\" as given in the input.
1034 \"starValues\" as given in the input.
1035
1036 \"freeIonSymbols\" which are the symbols used in the intermediate
    coupling basis.
1037 \"truncationEnergy\" which is the energy used to truncate the
    Hamiltonian.
1038 \"numE\" which is the number of electrons in the f^numE configuration
    .
1039 \"expData\" which is the experimental data used for fitting.
1040 \"problemVars\" which are the symbols considered for fitting
1041
1042 \"maxIterations\" which is the maximum number of iterations used by
    NMinimize.
1043 \"hamDim\" which is the dimension of the full Hamiltonian.
1044 \"allVars\" which are all the symbols defining the Hamiltonian under
    the aggregate simplifications.
1045 \"freeBies\" which are the free-ion parameters used to define the
    intermediate coupling basis.
1046 \"truncatedDim\" which is the dimension of the truncated Hamiltonian.
1047 \"compiledIntermediateFname\" the file name of the compiled function
    used for the truncated Hamiltonian.
1048
1049 \"fittedLevels\" which is the number of levels fitted for.
1050 \"actualSteps\" the number of steps that FindMiniminum actually took.
1051 \"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.
1052 \"rmsHistory\" which is a list of the \[Sigma] values found during
    the fitting.
1053 \"Appendix\" which is an association appended to the log file under
    the key \"Appendix\".
1054 \"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.
1055
1056 \"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.
1057 \"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.
1058
1059 The function admits the following options with default values:
1060   \"MaxHistory\" : determines how long the logs for the solver can be
      .
1061   \"MaxIterations\": determines the maximum number of iterations used
       by NMinimize.
1062   \"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\".
1063   \"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.
1064
1065   \"AccuracyGoal\": the accuracy goal used by NMinimize, default of
      5.
1066   \"TrucationEnergy\": if Automatic then the maximum energy in
      expData is taken, else it takes the value set by this option. In
```

```
    all cases the energies in expData are only considered up to this
    value.
  \"PrintFun\": the function used to print progress messages, the
    default is PrintTemporary.

  \"SlackChannel\": name of the Slack channel to which to dump
    progress messaages, the default is None which disables this option
    .
  \"ProgressView\": whether or not a progress window will be opened
    to show the progress of the solver, the default is True.
  \"SignatureCheck\": if True then then the function returns
    prematurely, returning a list with the symbols that would have
    defined the Hamiltonian after all simplifications have been
    applied. Useful to check the entire parameter set that the
    Hamiltonian has, which has to match one-to-one what is provided by
     startingValues.
  \"SaveEigenvectors\": if True then the both the eigenvectors and
    eigenvalues are saved under the \"states\" key of the returned
    association. If False then only the energies are saved, the
    default is False.

  \"AppendToLogFile\": an association appended to the log file under
    the key \"Appendix\".
  \"MagneticSimplifier\": a list of replacement rules to simplify the
     Marvin and pesudo-magnetic paramters. Here the ratios of the
    Marvin parameters and the pseudo-magnetic parameters are defined
    to simplify the magnetic part of the Hamiltonian.
  \"MagFieldSimplifier\": a list of replacement rules to specify a
    magnetic field (in T), if set to {}, then {Bx, By, Bz} can also be
     used as variables to be fitted for.

  \"SymmetrySimplifier\": a list of replacements rules to simplify
    the crystal field.
  \"OtherSimplifier\": an additional list of replacement rules that
    are applied to the Hamiltonian before computing with it. Here the
    spin-spin contribution can be turned off by setting \[Sigma]SS->0,
     which is the default.
";
Options[ClassicalFit] = {
  "MaxHistory"       -> 200,
  "MaxIterations"    -> 100,
  "FilePrefix"       -> "calcs",
  "ProgressView"     -> True,
  "TruncationEnergy" -> Automatic,
  "AccuracyGoal"     -> 5,
  "PrintFun"         -> PrintTemporary,
  "SlackChannel"     -> None,
  "ProgressView"     -> True,
  "SignatureCheck"   -> False,
  "AddConstantShift" -> False,
  "SaveEigenvectors" -> False,
  "AppendToLogFile"  -> <||>,
  "MagneticSimplifier" -> {
    M2 -> 56/100 M0,
    M4 -> 31/100 M0,
    P4 -> 1/2 P2,
    P6 -> 1/10 P2
    },
  "MagFieldSimplifier" -> {
    Bx -> 0,
    By -> 0,
    Bz -> 0
    },
  "SymmetrySimplifier" -> {
    B12->0, B14->0, B16->0, B34->0, B36->0, B56->0,
    S12->0 ,S14->0, S16->0, S22->0, S24->0, S26->0,
    S34->0 ,S36->0, S44->0, S46->0, S56->0, S66->0
    },
  "OtherSimplifier" -> {
    F0->0,
    P0->0,
    \[Sigma]SS->0,
    T11p->0, T11->0, T12->0, T14->0, T15->0,
    T16->0, T18->0, T17->0, T19->0, T2p->0
    },
  "ThreeBodySimplifier" -> <|
```

```
    1  -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0},
    2 -> {
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      },
    3  -> {},
    4  -> {},
    5  -> {},
    6  -> {},
    7  -> {},
    8  -> {},
    9  -> {},
    10 -> {},
    11 -> {},
    12 -> {
      T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      },
    13->{
      T2->0, T3->0, T4->0, T6->0, T7->0, T8->0,
      T11p->0, T11->0, T12->0, T14->0, T15->0, T16->0, T18->0, T17
    ->0, T19->0,
      T2p->0
      }
    |>,
    "FreeIonSymbols" -> {F0, F2, F4, F6, ζ}
};
ClassicalFit[numE_Integer, expData_List, excludeDataIndices_List,
    problemVars_List, startValues_Association, \[Sigma]exp_?NumericQ,
    constraints_List, OptionsPattern[]]:=Module[
  {accuracyGoal, activeVarIndices, activeVars, activeVarsString,
    activeVarsWithRange, allFreeEnergies, allFreeEnergiesSorted,
    allVars, allVarsVec, argsForEvalInsideOfTheIntermediateSystems,
    argsOfTheIntermediateEigensystems, aVar, aVarPosition, basis,
    basisChanger, basisChangerBlocks, bestError, bestParams, bestRMS,
    blockShifts, blockSizes, colIdx, compiledDiagonal,
    compiledIntermediateFname, constrainedProblemVars,
    constrainedProblemVarsList, covMat, currentRMS, degressOfFreedom,
    dependentVars, diagonalBlocks, diagonalScalarBlocks, diff,
    eigenEnergies, eigenvalueDispenserTemplate, eigenVectors,
    elevatedIntermediateEigensystems, endTime, fmSol, fmSolAssoc,
    fractionalWidth, freeBies, freeIenergiesAndMultiplets,
    freeionSymbols, fullHam, fullSolVec, funcString, ham, hamDim,
    hamEigenvaluesTemplate, hamString, hess, indepSolVecVec, indepVars
    , intermediateHam, isolationValues, jobVars, lin, linMat, ln,
    lnParams, logFilePrefix, logFname, magneticSimplifier,
    maxFreeEnergy, maxHistory, maxIterations, methodString,
    methodStringTemplate, minFreeEnergy, minpoly, modelSymbols,
    multipletAssignments, needlePosition, numBlocks, numQSignature,
    numReps, solCompendium, openNotebooks, ordering, othersFixed,
    otherSimplifier, p0, paramBest, paramSigma, perHam, polySols,
    presentDataIndices, PrintFun, problemVarsPositions, problemVarsQ,
    problemVarsQString, problemVarsVec, problemVarsWithStartValues,
    reducedModelSymbols, resultMessage, roundedTruncationEnergy,
    rowIdx, runningInteractive, shiftToggle, simplifier, slackChan,
    sol, solAssoc, sols, solWithUncertainty, sortedTruncationIndex,
    sqdiff, standardValues, starTime, startingValues, startTime,
    startVarValues, states, steps, symmetrySimplifier,
    theIntermediateEigensystems, TheIntermediateEigensystems,
    TheTruncatedAndSignedPathGenerator, thisPoly, threadHeaderTemplate
    , threadMessage, threadTS, timeTaken, totalVariance,
    truncadedFname, truncatedIntermediateBasis,
    truncatedIntermediateHam, truncationEnergy, truncationIndices,
    truncationUmbral, usingInitialRange, varHash, varIdx,
    varsWithConstants, varWithValsSignature, \[Lambda]0Vec, \[Lambda]
    exp},
  (
    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"];
    symmetrySimplifier  = OptionValue["SymmetrySimplifier"];
    otherSimplifier     = OptionValue["OtherSimplifier"];
    threeBodySimplifier = If[Head[OptionValue["ThreeBodySimplifier"]]
    == Association,
                           OptionValue["ThreeBodySimplifier"][numE],
                           OptionValue["ThreeBodySimplifier"]
                          ];

    truncationEnergy = If[OptionValue["TruncationEnergy"]===Automatic
    ,
      PrintFun["Truncation energy set to Automatic, using the maximum
     energy in the data ..."];
      Max[Select[First /@ expData, NumericQ[#] &]],
      OptionValue["TruncationEnergy"]
      ];
    PrintFun["Using a truncation energy of ", truncationEnergy, " K"
    ];

    simplifier          = Join[magneticSimplifier,
                          magFieldSimplifier,
                          symmetrySimplifier,
                          threeBodySimplifier,
                          otherSimplifier];

    PrintFun["Determining gaps in the data ..."];
    (* the indices that are numeric in expData whatever is non-
    numeric is assumed as a known gap *)
    presentDataIndices = Flatten[Position[expData, {_?(NumericQ[#] &)
    , ___}]];
    (* some indices omitted here based on the excludeDataIndices
    argument *)
    presentDataIndices = Complement[presentDataIndices,
    excludeDataIndices];

    hamDim                          = Binomial[14, numE];
    solCompendium["simplifier"]          = simplifier;
    solCompendium["excludeDataIndices"] = excludeDataIndices;
    solCompendium["startValues"]         = startValues;
    solCompendium["freeIonSymbols"]      = freeIonSymbols;
    solCompendium["truncationEnergy"]    = truncationEnergy;
    solCompendium["numE"]                = numE;
    solCompendium["expData"]             = expData;
    solCompendium["problemVars"]         = problemVars;
    solCompendium["maxIterations"]       = maxIterations;
    solCompendium["hamDim"]              = hamDim;
    solCompendium["constraints"]         = constraints;
    modelSymbols  = Sort[Select[paramSymbols, Not[MemberQ[Join[
    racahSymbols, juddOfeltIntensitySymbols, chenSymbols,{t2Switch, \[
    Epsilon],gs}],#]]&]];
    (* remove the symbols that will be removed by the simplifier, no
    symbol should remain here that is not in the symbolic Hamiltonian
    *)
    reducedModelSymbols = Select[modelSymbols, Not[MemberQ[Keys[
    simplifier],#]]&];

    (* this is useful to understand what are the arguments of the
    truncated compiled Hamiltonian *)
    If[OptionValue["SignatureCheck"],
      (
        Print["Given the model parameters and the simplifying
    assumptions, the resultant model parameters are:"];
```

```
1215          Print[{reducedModelSymbols}];
1216          Print["Exiting ..."];
1217          Return[""];
1218        )
1219      ];
1220
1221      (* calculate the basis *)
1222      PrintFun["Retrieving the LSJMJ basis for f^", numE, " ..."];
1223      basis    = BasisLSJMJ[numE];
1224
1225      (* get the reference parameters from LaF3 *)
1226      PrintFun["Getting reference free-ion parameters for ",ln," using
        LaF3 ..."];
1227      lnParams = LoadParameters[ln];
1228      freeBies = Prepend[Values[(#->(#/.lnParams)) &/@ freeIonSymbols],
          numE];
1229      (* a more explicit alias *)
1230      allVars           = reducedModelSymbols;
1231      numericConstraints = Association@Select[constraints, NumericQ
        [#[[2]]] &];
1232      standardValues    = allVars /. Join[lnParams, numericConstraints
        ];
1233      solCompendium["allVars"]  = allVars;
1234      solCompendium["freeBies"] = freeBies;
1235
1236      (* reload compiled version if found *)
1237      varHash                   = Hash[{numE, allVars, freeBies,
        truncationEnergy, simplifier}];
1238      compiledIntermediateFname = ln<>"-compiled-intermediate-truncated
        -ham-"<>ToString[varHash]<>".mx";
1239      compiledIntermediateFname = FileNameJoin[{moduleDir, "compiled",
        compiledIntermediateFname}];
1240      solCompendium["compiledIntermediateFname"] =
        compiledIntermediateFname;
1241
1242      If[FileExistsQ[compiledIntermediateFname],
1243        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 ..."];
1244        PrintFun["Using : ", compiledIntermediateFname];
1245        {compileIntermediateTruncatedHam, truncatedIntermediateBasis} =
         Import[compiledIntermediateFname];,
1246      (
1247        (* grab the Hamiltonian preserving its block structure *)
1248        PrintFun["Assembling the Hamiltonian for f^",numE," keeping the
         block structure ..."];
1249        ham          = HamMatrixAssembly[numE, "ReturnInBlocks"->True];
1250        (* apply the simplifier *)
1251        PrintFun["Simplifying using the aggregate set of simplification
         rules ..."];
1252        ham          = Map[ReplaceInSparseArray[#, simplifier]&, ham,
        {2}];
1253        PrintFun["Zeroing out every symbol in the Hamiltonian that is
        not a free-ion parameter ..."];
1254        (* Get the free ion symbols *)
1255        freeIonSimplifier = (#->0) & /@ Complement[reducedModelSymbols,
         freeIonSymbols];
1256        (* Take the diagonal blocks for the intermediate analysis *)
1257        PrintFun["Grabbing the diagonal blocks of the Hamiltonian ..."
        ];
1258        diagonalBlocks       = Diagonal[ham];
1259        (* simplify them to only keep the free ion symbols *)
1260        PrintFun["Simplifying the diagonal blocks to only keep the free
         ion symbols ..."];
1261        diagonalScalarBlocks = ReplaceInSparseArray[#,freeIonSimplifier
        ]&/@diagonalBlocks;
1262        (* these include the MJ quantum numbers, remove that *)
1263        PrintFun["Contracting the basis vectors by removing the MJ
        quantum numbers from the diagonal blocks ..."];
1264        diagonalScalarBlocks = FreeHam[diagonalScalarBlocks, numE];
1265
1266        argsOfTheIntermediateEigensystems         = StringJoin[Riffle[
        Prepend[(ToString[#]<>"v_") & /@ freeIonSymbols,"numE_"],", "]];
1267        argsForEvalInsideOfTheIntermediateSystems = StringJoin[Riffle[(
        ToString[#]<>"v") & /@ freeIonSymbols,", "]];
```

```
1268      PrintFun["argsOfTheIntermediateEigensystems = ",
          argsOfTheIntermediateEigensystems];
1269      PrintFun["argsForEvalInsideOfTheIntermediateSystems = ",
          argsForEvalInsideOfTheIntermediateSystems];
1270      PrintFun["(if the following fails, it might help to see if the
          arguments of TheIntermediateEigensystems match the ones shown
          above)"];
1271
1272      (* 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 *)
1273      (* compile one function for each of the blocks *)
1274      PrintFun["Compiling functions for the diagonal blocks of the
          Hamiltonian ..."];
1275      compiledDiagonal = Compile[Evaluate[freeIonSymbols], Evaluate[N
          [Normal[#]]]]&/@diagonalScalarBlocks;
1276      (* use that to create a function that will calculate the free-
          ion eigensystem *)
1277      TheIntermediateEigensystems[numEv_, F0v_, F2v_, F4v_, F6v_, ζv_
          ] := (
1278        theNumericBlocks = (#[F0v, F2v, F4v, F6v, ζv]&)/
          @compiledDiagonal;
1279        theIntermediateEigensystems = Eigensystem/@theNumericBlocks;
1280        Js      = AllowedJ[numEv];
1281        basisJ = BasisLSJMJ[numEv,"AsAssociation"->True];
1282        (* having calculated the eigensystems with the removed
          degeneracies, put the degeneracies back in explicitly *)
1283        elevatedIntermediateEigensystems = MapIndexed[EigenLever[#1,2
          Js[[#2[[1]]]]+1]&, theIntermediateEigensystems];
1284        (* Identify a single MJ to keep *)
1285        pivot = If[EvenQ[numEv],0,-1/2];
1286        LSJmultiplets = (#[[1]]<>ToString[InputForm[#[[2]]]])&/
          @Select[BasisLSJMJ[numEv],#[[-1]]== pivot &];
1287        (* calculate the multiplet assignments that the intermediate
          basis eigenvectors have *)
1288        needlePosition = 0;
1289        multipletAssignments = Table[
1290          (
1291            J          = Js[[idx]];
1292            eigenVecs = theIntermediateEigensystems[[idx]][[2]];
1293            majorComponentIndices         = Ordering[Abs[#]][[-1]]&/
          @eigenVecs;
1294            majorComponentIndices        += needlePosition;
1295            needlePosition               += Length[
          majorComponentIndices];
1296            majorComponentAssignments     = LSJmultiplets[[#]]&/
          @majorComponentIndices;
1297            (* All of the degenerate eigenvectors belong to the same
          multiplet*)
1298            elevatedMultipletAssignments = ListRepeater[
          majorComponentAssignments,2J+1];
1299            elevatedMultipletAssignments
1300          ),
1301        {idx, 1, Length[Js]}
1302        ];
1303        (* put together the multiplet assignments and the energies *)
1304        freeIenergiesAndMultiplets = Transpose/@Transpose[{First/
          @elevatedIntermediateEigensystems, multipletAssignments}];
1305        freeIenergiesAndMultiplets = Flatten[
          freeIenergiesAndMultiplets, 1];
1306        (* calculate the change of basis matrix using the
          intermediate coupling eigenvectors *)
1307        basisChanger = BlockDiagonalMatrix[Transpose/@Last/
          @elevatedIntermediateEigensystems];
1308        basisChanger = SparseArray[basisChanger];
1309        Return[{theIntermediateEigensystems, multipletAssignments,
          elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
          basisChanger}]
1310      );
1311
1312      PrintFun["Calculating the intermediate eigensystems for ",ln,"
          using free-ion params from LaF3 ..."];
1313      (* calculate intermediate coupling basis using the free-ion
          params for LaF3 *)
1314      {theIntermediateEigensystems, multipletAssignments,
          elevatedIntermediateEigensystems, freeIenergiesAndMultiplets,
```

```
    basisChanger} = TheIntermediateEigensystems@@freeBies;
1315
1316    (* use that intermediate coupling basis to compile a function
    for the full Hamiltonian *)
1317    allFreeEnergies = Flatten[First/
    @elevatedIntermediateEigensystems];
1318    (* important that the intermediate coupling basis have attached
     energies, which make possible the truncation *)
1319    ordering = Ordering[allFreeEnergies];
1320    (* sort the free ion energies and determine which indices
    should be included in the truncation *)
1321    allFreeEnergiesSorted          = Sort[allFreeEnergies];
1322    {minFreeEnergy, maxFreeEnergy} = MinMax[allFreeEnergies];
1323    (* determine the index at which the energy is equal or larger
    than the truncation energy *)
1324    sortedTruncationIndex = Which[
1325      truncationEnergy > (maxFreeEnergy-minFreeEnergy),
1326      hamDim,
1327      True,
1328      FirstPosition[allFreeEnergiesSorted-Min[allFreeEnergiesSorted
    ],x_/;x>truncationEnergy,{0},1][[1]]
1329    ];
1330    (* the actual energy at which the truncation is made *)
1331    roundedTruncationEnergy = allFreeEnergiesSorted[[
    sortedTruncationIndex]];
1332
1333    (* the indices that enact the truncation *)
1334    truncationIndices = ordering[[;;sortedTruncationIndex]];
1335    (* Return[{basisChanger, ham, truncationIndices}]; *)
1336    PrintFun["Computing the block structure of the change of basis
    array ..."];
1337    blockSizes = BlockArrayDimensionsArray[ham];
1338    basisChangerBlocks = ArrayBlocker[basisChanger, blockSizes];
1339    blockShifts = First /@ Diagonal[blockSizes];
1340    numBlocks = Length[blockSizes];
1341    (* using the ham (with all the symbols) change the basis to the
     computed one *)
1342    PrintFun["Changing the basis of the Hamiltonian to the
    intermediate coupling basis ..."];
1343    (* intermediateHam            = Transpose[basisChanger].ham.
    basisChanger; *)
1344    (* Return[{basisChangerBlocks, ham}]; *)
1345    intermediateHam = BlockMatrixMultiply[ham, basisChangerBlocks];
1346    PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
1347    Do[
1348      intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1349      {rowIdx, 1, numBlocks},
1350      {colIdx, 1, numBlocks}
1351    ];
1352    intermediateHam = BlockMatrixMultiply[BlockTranspose[
    basisChangerBlocks],intermediateHam];
1353    PrintFun["Distributing products inside of symbolic matrix
    elements to keep complexity in check ..."];
1354    Do[
1355      intermediateHam[[rowIdx, colIdx]] = MapToSparseArray[
    intermediateHam[[rowIdx, colIdx]], Distribute /@ # &];,
1356      {rowIdx, 1, numBlocks},
1357      {colIdx, 1, numBlocks}
1358    ];
1359    (* using the truncation indices truncate that one *)
1360    PrintFun["Truncating the Hamiltonian ..."];
1361    truncatedIntermediateHam = TruncateBlockArray[intermediateHam,
    truncationIndices, blockShifts];
1362    (* these are the basis vectors for the truncated hamiltonian *)
1363    PrintFun["Saving the truncated intermediate basis ..."];
1364    truncatedIntermediateBasis = basisChanger[[All,
    truncationIndices]];
1365
1366    PrintFun["Compiling a function for the truncated Hamiltonian
    ..."];
1367    (* compile a function that will calculate the truncated
    Hamiltonian given the parameters in allVars, this is the function
    to be use in fitting *)
1368    compileIntermediateTruncatedHam = Compile[Evaluate[allVars],
```

```
    Evaluate[truncatedIntermediateHam]];
1369    (* save the compiled function *)
1370    PrintFun["Saving the compiled function for the truncated
    Hamiltonian and the truncated intermediate basis ..."];
1371    Export[compiledIntermediateFname, {
    compileIntermediateTruncatedHam, truncatedIntermediateBasis}];
1372    )
1373    ];
1374
1375    truncationUmbral = Dimensions[truncatedIntermediateBasis][[2]];
1376    PrintFun["The truncated Hamiltonian has a dimension of ",
    truncationUmbral, "x", truncationUmbral, " ..."];
1377    presentDataIndices = Select[presentDataIndices, # <=
    truncationUmbral &];
1378    solCompendium["presentDataIndices"] = presentDataIndices;
1379
1380    (* the problemVars are the symbols that will be fitted for *)
1381
1382    PrintFun["Starting up the fitting process using the Levenberg-
    Marquardt method ..."];
1383    (* using the problemVars I need to create the argument list
    including _?NumericQ *)
1384    problemVarsQ = (ToString[#] <> "_?NumericQ") & /@ problemVars;
1385    problemVarsQString = StringJoin[Riffle[problemVarsQ, ", "]];
1386    (* we also need to have the padded arguments with the variables
    in the right position and the fixed values in the remaining ones
    *)
1387    problemVarsPositions = Position[allVars, #][[1, 1]] & /@
    problemVars;
1388    problemVarsString = StringJoin[Riffle[ToString /@ problemVars, ",
     "]];
1389    (* 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 *)
1390    varsWithConstants = standardValues;
1391    varsWithConstants[[problemVarsPositions]] = problemVars;
1392    varsWithConstantsString = ToString[varsWithConstants];
1393
1394    (* this following function serves eigenvalues from the
    Hamiltonian, has memoization so it might grow to use a lot of RAM
    *)
1395    Clear[HamSortedEigenvalues];
1396    hamEigenvaluesTemplate = StringTemplate["
1397    HamSortedEigenvalues[`problemVarsQ`]:=(
1398      ham          = compileIntermediateTruncatedHam@@`
    varsWithConstants`;
1399      eigenValues = Sort@Eigenvalues@ham;
1400      eigenValues = eigenValues - Min[eigenValues];
1401      HamSortedEigenvalues[`problemVarsString`] = eigenValues;
1402      Return[eigenValues]
1403    )"];
1404    hamString = hamEigenvaluesTemplate[<|
1405        "problemVarsQ" -> problemVarsQString,
1406        "varsWithConstants" -> varsWithConstantsString,
1407        "problemVarsString" -> problemVarsString
1408        |>];
1409    ToExpression[hamString];
1410
1411    (* 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 *)
1412    eigenvalueDispenserTemplate = StringTemplate["
1413    PartialHamEigenvalues[`problemVarsQ`][i_]:=(
1414      eigenVals = HamSortedEigenvalues[`problemVarsString`];
1415      eigenVals[[i]]
1416    )
1417    "];
1418    eigenValueDispenserString =
1419      eigenvalueDispenserTemplate[<|
1420        "problemVarsQ"      -> problemVarsQString,
1421        "problemVarsString" -> problemVarsString
1422        |>];
1423    ToExpression[eigenValueDispenserString];
1424
1425    PringFun["Determining the free variables after constraints ..."];
1426    constrainedProblemVars = (problemVars /. constraints);
```

```
1427      constrainedProblemVarsList = Variables[constrainedProblemVars];
1428      If[addShift,
1429        PrintFun["Adding a constant shift to the fitting parameters ...
          "];
1430        constrainedProblemVarsList = Append[constrainedProblemVarsList,
           \[Epsilon]]
1431      ];
1432
1433      indepVars = Complement[problemVars, #[[1]] & /@ constraints];
1434      stringPartialVars = ToString/@constrainedProblemVarsList;
1435
1436      paramSols  = {};
1437      rmsHistory = {};
1438      steps      = 0;
1439      problemVarsWithStartValues = KeyValueMap[{#1,#2} &, startValues];
1440      If[addShift,
1441        problemVarsWithStartValues = Append[problemVarsWithStartValues,
           {\[Epsilon] ,0}];
1442      ];
1443      openNotebooks     = If[runningInteractive,
1444                ("WindowTitle"/.NotebookInformation[#]) & /@ Notebooks
          [],
1445                {}];
1446      If[Not[MemberQ[openNotebooks,"Solver Progress"]] && OptionValue["
          ProgressView"],
1447        ProgressNotebook["Basic"->False]
1448      ];
1449      degressOfFreedom = Length[presentDataIndices] - Length[
          problemVars] - 1;
1450      PrintFun["Fitting for ", Length[presentDataIndices], " data
          points with ", Length[problemVars], " free parameters.", " The
          effective degrees of freedom are ", degressOfFreedom, " ..."];
1451
1452      PrintFun["Starting the fitting process ..."];
1453      startTime=Now;
1454      shiftToggle = If[addShift, 1, 0];
1455      sol = FindMinimum[
1456        Sum[(expData[[j]][[1]] - (PartialHamEigenvalues @@
          constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2,
1457        {j, presentDataIndices}],
1458        problemVarsWithStartValues,
1459        Method        -> "LevenbergMarquardt",
1460        MaxIterations -> OptionValue["MaxIterations"],
1461        AccuracyGoal  -> OptionValue["AccuracyGoal"],
1462        StepMonitor :> (
1463          steps      += 1;
1464          currentRMS = Sum[(expData[[j]][[1]] - (PartialHamEigenvalues
          @@ constrainedProblemVars)[j] - shiftToggle * \[Epsilon])^2, {j,
          presentDataIndices}];
1465          currentRMS = Sqrt[currentRMS / degressOfFreedom];
1466          paramSols  = AddToList[paramSols, constrainedProblemVarsList,
           maxHistory];
1467          rmsHistory = AddToList[rmsHistory, currentRMS, maxHistory];
1468          )
1469        ];
1470      endTime = Now;
1471      timeTaken = QuantityMagnitude[endTime - startTime, "Seconds"];
1472      PrintFun["Solution found in ", timeTaken, "s"];
1473
1474      solVec = constrainedProblemVars /. sol[[-1]];
1475      indepSolVec = indepVars /. sol[[-1]];
1476      If[addShift,
1477        \[Epsilon]Best = \[Epsilon]/. sol[[-1]],
1478        \[Epsilon]Best = 0
1479      ];
1480      fullSolVec = standardValues;
1481      fullSolVec[[problemVarsPositions]] = solVec;
1482      PrintFun["Calculating the numerical Hamiltonian corresponding to
          the solution ..."];
1483      fullHam = compileIntermediateTruncatedHam @@ fullSolVec;
1484      PrintFun["Calculating energies and eigenvectors ..."];
1485      {eigenEnergies, eigenVectors} = Eigensystem[fullHam];
1486      states = Transpose[{eigenEnergies, eigenVectors}];
1487      states = SortBy[states, First];
1488      eigenEnergies = First /@ states;
1489      PrintFun["Shifting energies to make ground state zero of energy
```

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

PrintFun["Calculating the eigenvalue vector at solution ..."];
\[Lambda]0Vec = Transpose[{eigenEnergies[[presentDataIndices]]}];
PrintFun["Putting together the experimental vector ..."];
\[Lambda]exp = Transpose[{First /@ expData[[presentDataIndices
]]}];
problemVarsVec = If[addShift,
  Transpose[{constrainedProblemVarsList[[;;-2]]}],
  Transpose[{constrainedProblemVarsList}]
];
indepSolVecVec = Transpose[{indepSolVec}];
PrintFun["Calculating the difference between eigenvalues at
solution ..."];
diff = If[linMat=={},
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best,
  (\[Lambda]0Vec - \[Lambda]exp)  + \[Epsilon]Best + linMat[[
presentDataIndices]].(problemVarsVec - indepSolVecVec)
];
PrintFun["Calculating the sum of squares of differences around
solution ... "];
sqdiff        = Expand[(Transpose[diff] . diff)[[1, 1]]];
PrintFun["Calculating the minimum (which should coincide with sol
 ..."];
minpoly       = sqdiff /. AssociationThread[problemVars -> solVec
];
fmSolAssoc    = Association[sol[[2]]];
totalVariance = Length[presentDataIndices]*\[Sigma]exp^2;
PrintFun["Calculating the uncertainty in the parameters ..."];
solWithUncertainty = Table[
  (
    aVar        = constrainedProblemVarsList[[varIdx]];
    paramBest   = aVar /. fmSolAssoc;
    othersFixed = AssociationThread[Delete[
constrainedProblemVarsList[[;;tail]], varIdx] -> Delete[
indepSolVec, varIdx]];
    thisPoly    = sqdiff /. othersFixed;
    polySols    = Last /@ Last /@ Solve[thisPoly == minpoly + 1*
totalVariance];
    polySols    = Select[polySols, Im[#] == 0 &];
    paramSigma  = Max[polySols] - Min[polySols];
    (aVar -> {paramBest, paramSigma})
  ),
{varIdx, 1, Length[constrainedProblemVarsList]-shiftToggle}
];
```

```
    PrintFun["Calculating the covariance matrix ..."];
    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;
```

```
6
7  (* Spectroscopic niceties*)
8  theLanthanides = {"Ce", "Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy
       ", "Ho", "Er", "Tm", "Yb"};
9  theActinides   = {"Ac", "Th", "Pa", "U", "Np", "Pu", "Am", "Cm", "Bk"
       , "Cf", "Es", "Fm", "Md", "No", "Lr"};
10 theTrivalents  = {"Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho
       ", "Er", "Tm"};
11 specAlphabet   = "SPDFGHIKLMNOQRTUV"
12
13 (* SI *)
14 hPlanck = 6.62607015 * 10^-34;          (* Planck's constant in J s
       *)
15 hBar    = hPlanck    / (2 \[Pi]);       (* reduced Planck's constant
       in J s *)
16 \[Mu]B  = 9.2740100783 * 10^-24;        (* Bohr magneton in SI *)
17 me      = 9.1093837015 * 10^-31;        (* electron mass in kg *)
18 cLight  = 2.99792458   * 10^8;          (* speed of light in m/s *)
19 eCharge = 1.602176634  * 10^-19;        (* elementary charge in SI *)
20 \[Epsilon]0  = 8.8541878128 * 10^-12;   (* electric permittivity in
       vacuum in SI *)
21 \[Mu]0  = 4 \[Pi] * 10^-7;              (* magnetic permeability in
       vacuum in SI *)
22 αFine = 1/137.036;                (* fine structure constant *)
23
24 bohrRadius    = 5.29177*10^-11;         (* Bohr radius in m *)
25 hartreeEnergy = hBar^2 / (me * bohrRadius^2); (* Hartree energy in J
       *)
26 hartreeTime   = hBar / hartreeEnergy; (* Hartree time in s *)
27
28 (* Hartree atomic units *)
29 hPlanckHartree = 2 \[Pi]; (* Planck's constant in Hartree *)
30 meHartree      = 1;       (* electron mass in Hartree *)
31 cLightHartree  = 137.036; (* speed of light in Hartree *)
32 eChargeHartree = 1;       (* elementary charge in Hartree *)
33 \[Mu]0Hartree  = αFine^2; (* magnetic permeability in vacuum in
       Hartree *)
34
35 (* some conversion factors *)
36 eVToJoule       = eCharge;
37 jouleToeV       = 1 / eVToJoule;
38 jouleToHartree  = 1 / hartreeEnergy;
39 eVToKayser      = eCharge /( hPlanck * cLight * 100 ); (* 1 eV =
       8065.54429 cm^-1 *)
40 kayserToeV      = 1 / eVToKayser;
41 teslaToKayser   = 2 * \[Mu]B / hPlanck / cLight / 100;
42 kayserToHartree = kayserToeV * eVToJoule * jouleToHartree;
43 hartreeToKayser = 1 / kayserToHartree;
44
45 EndPackage[];
```

12.4 qplotter.m

This module has a few useful plotting routines.

```
1
2  BeginPackage["qplotter`"];
3
4  GetColor;
5  IndexMappingPlot;
6  ListLabelPlot;
7  AutoGraphicsGrid;
8  SpectrumPlot;
9  WaveToRGB;
10
11 Begin["`Private`"];
12
13   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.";
14   Options[AutoGraphicsGrid] = Options[GraphicsGrid];
15   AutoGraphicsGrid[graphsList_, opts : OptionsPattern[]] :=
16     (
17       numGraphs = Length[graphsList];
18       width = Floor[Sqrt[numGraphs]];
```

```
      height = Ceiling[numGraphs/width];
      groupedGraphs = Partition[graphsList, width, width, 1, Null];
      GraphicsGrid[groupedGraphs, opts]
    )

Options[IndexMappingPlot] = Options[Graphics];
IndexMappingPlot::usage =
  "IndexMappingPlot[pairs] take a list of pairs of integers and
  creates a visual representation of how they are paired. The first
  indices being depicted in the bottom and the second indices being
  depicted on top.";
IndexMappingPlot[pairs_, opts : OptionsPattern[]] := Module[{width,
   height}, (
  width = Max[First /@ pairs];
  height = width/3;
  Return[
    Graphics[{{Tooltip[Point[{#[[1]], 0}],#[[1]]]}, Tooltip[Point
  [{#[[2]], height}],#[[2]]],
        Line[{{#[[1]], 0}, {#[[2]], height}}]} & /@ pairs, opts,
  ImageSize -> 800]]
  )
  ]

TickCompressor[fTicks_] :=
Module[{avgTicks, prevTickLabel, groupCounter, groupTally, idx,
  tickPosition, tickLabel, avgPosition, groupLabel}, (avgTicks =
  {};
  prevTickLabel = fTicks[[1, 2]];
  groupCounter = 0;
  groupTally = 0;
  idx = 1;
  Do[({tickPosition, tickLabel} = tick;
    If[
    tickLabel === prevTickLabel,
    (groupCounter += 1;
      groupTally += tickPosition;
      groupLabel = tickLabel;),
    (
      avgPosition = groupTally/groupCounter;
      avgTicks = Append[avgTicks, {avgPosition, groupLabel}];
      groupCounter = 1;
      groupTally = tickPosition;
      groupLabel = tickLabel;
      )
    ];
    If[idx != Length[fTicks],
    prevTickLabel = tickLabel;
    idx += 1;]
    ), {tick, fTicks}];
  If[Or[Not[prevTickLabel === tickLabel], groupCounter > 1],
  (
    avgPosition = groupTally/groupCounter;
    avgTicks = Append[avgTicks, {avgPosition, groupLabel}];
    )
  ];
  Return[avgTicks];)]

GetColor[s_Style] := s /. Style[_, c_] :> c
GetColor[_] := Black

ListLabelPlot::usage="ListLabelPlot[data, labels] takes a list of
  numbers with corresponding labels. The data is grouped according
  to the labels and a ListPlot is created with them so that each
  group has a different color and their corresponding label is shown
   in the horizontal axis.";
Options[ListLabelPlot] = Join[Options[ListPlot], {"TickCompression"
  ->True,
"LabelLevels"->1}];
ListLabelPlot[data_, labels_, opts : OptionsPattern[]] := Module[
  {uniqueLabels, pallete, groupedByTerm, groupedKeys, scatterGroups
  ,
  groupedColors, frameTicks, compTicks, bottomTicks, topTicks},
  (
  uniqueLabels  = DeleteDuplicates[labels];
  pallete = Table[ColorData["Rainbow", i], {i, 0, 1,
      1/(Length[uniqueLabels] - 1)}];
```

```
    uniqueLabels  = (#[[1]] -> #[[2]]) & /@ Transpose[{RandomSample[
    uniqueLabels], pallete}];
    uniqueLabels  = Association[uniqueLabels];
    groupedByTerm = GroupBy[Transpose[{labels, Range[Length[data]],
    data}], First];
    groupedKeys   = Keys[groupedByTerm];
    scatterGroups = Transpose[Transpose[#][[2 ;; 3]]] & /@ Values[
    groupedByTerm];
    groupedColors = uniqueLabels[#] & /@ groupedKeys;
    frameTicks    = {Transpose[{Range[Length[data]],
      Style[Rotate[#, 90 Degree], uniqueLabels[#]] & /@ labels}],
      Automatic};
      If[OptionValue["TickCompression"], (
          compTicks = TickCompressor[frameTicks[[1]]];
          bottomTicks =
              MapIndexed[
              If[EvenQ[First[#2]], {#1[[1]],
                  Tooltip[Style["\[SmallCircle]", GetColor
    [#1[[2]]]],#1[[2]]]
                  }, #1] &, compTicks];
          topTicks =
              MapIndexed[
              If[OddQ[First[#2]], {#1[[1]],
                  Tooltip[Style["\[SmallCircle]", GetColor
    [#1[[2]]]],#1[[2]]]
                  }, #1] &, compTicks];
          frameTicks = {{Automatic, Automatic}, {bottomTicks,
    topTicks}};)
    ];
    ListPlot[scatterGroups,
      opts,
      Frame->True,
      AxesStyle -> {Directive[Black, Dotted], Automatic},
      PlotStyle -> groupedColors,
      FrameTicks -> frameTicks]
    )
    ]

  WaveToRGB::usage="WaveToRGB[wave, gamma] takes a wavelength in nm
    and returns the corresponding RGB color. The gamma parameter is
    optional and defaults to 0.8. The wavelength wave is assumed to be
     in nm. If the wavelength is below 380 the color will be the same
    as for 380 nm. If the wavelength is above 750 the color will be
    the same as for 750 nm. The function returns an RGBColor object.
    REF: https://www.noah.org/wiki/wave_to_rgb_in_Python. ";
  WaveToRGB[wave_, gamma_ : 0.8] := (
    wavelength = (wave);
    Which[
      wavelength < 380,
        wavelength = 380,
      wavelength > 750,
        wavelength = 750
    ];
    Which[380 <= wavelength <= 440,
    (
      attenuation = 0.3 + 0.7*(wavelength - 380)/(440 - 380);
      R = ((-(wavelength - 440)/(440 - 380))*attenuation)^gamma;
      G = 0.0;
      B = (1.0*attenuation)^gamma;
      ),
    440 <= wavelength <= 490,
    (
      R = 0.0;
      G = ((wavelength - 440)/(490 - 440))^gamma;
      B = 1.0;
      ),
    490 <= wavelength <= 510,
    (
      R = 0.0;
      G = 1.0;
      B = (-(wavelength - 510)/(510 - 490))^gamma;
      ),
    510 <= wavelength <= 580,
    (
      R = ((wavelength - 510)/(580 - 510))^gamma;
      G = 1.0;
```

```
      B = 0.0;
      ),
    580 <= wavelength <= 645,
    (
      R = 1.0;
      G = (-(wavelength - 645)/(645 - 580))^gamma;
      B = 0.0;
      ),
    645 <= wavelength <= 750,
    (
      attenuation = 0.3 + 0.7*(750 - wavelength)/(750 - 645);
      R = (1.0*attenuation)^gamma;
      G = 0.0;
      B = 0.0;
      ),
    True,
    (
      R = 0;
      G = 0;
      B = 0;
      )];
    Return[RGBColor[R, G, B]]
    )

  FuzzyRectangle::usage = "FuzzyRectangle[xCenter, width, ymin,
    height, color] creates a polygon with a fuzzy edge. The polygon is
     centered at xCenter and has a full horizontal width of width. The
     bottom of the polygon is at ymin and the height is height. The
    color of the polygon is color. The left edge and the right edge of
     the resulting polygon will be transparent and the middle will be
    colored. The polygon is returned as a list of polygons.";
  FuzzyRectangle[xCenter_, width_, ymin_, height_, color_, intensity_
    :1] := Module[
    {intenseColor, nocolor, ymax, polys},
  (
    nocolor = Directive[Opacity[0], color];
    ymax = ymin + height;
    intenseColor = Directive[Opacity[intensity], color];
    polys = {
      Polygon[{
        {xCenter - width/2, ymin},
        {xCenter, ymin},
        {xCenter, ymax},
        {xCenter - width/2, ymax}},
        VertexColors -> {
          nocolor,
          intenseColor,
          intenseColor,
          nocolor,
          nocolor}],
      Polygon[{
        {xCenter, ymin},
        {xCenter + width/2, ymin},
        {xCenter + width/2, ymax},
        {xCenter, ymax}},
        VertexColors -> {
          intenseColor,
          nocolor,
          nocolor,
          intenseColor,
          intenseColor}]
      };
    Return[polys]
    );
  ]

  Options[SpectrumPlot] = Options[Graphics];
  Options[SpectrumPlot] = Join[Options[SpectrumPlot], {"Intensities"
    -> {},"Tooltips" -> True, "Comments" -> {}, "SpectrumFunction" ->
    WaveToRGB}];
  SpectrumPlot::usage="SpectrumPlot[lines, widthToHeightAspect,
    lineWidth] takes a list of spectral lines and creates a visual
    representation of them. The lines are represented as fuzzy
    rectangles with a width of lineWidth and a height that is
    determined by the overall condition that the width to height ratio
     of the resulting graph is widthToHeightAspect. The color of the
```

```
    lines is determined by the wavelength of the line. The function
    assumes that the lines are given in nm.
  If the lineWidth parameter is a single number, then every line
    shares that width. If the lineWidth parameter is a list of numbers
    , then each line has a different width. The function returns a
    Graphics object. The function also accepts any options that
    Graphics accepts. The background of the plot is black by default.
    The plot range is set to the minimum and maximum wavelength of the
     given lines.
  Besides the options for Graphics the function also admits the
    option Intensities. This option is a list of numbers that
    determines the intensity of each line. If the Intensities option
    is not given, then the lines are drawn with full intensity. If the
     Intensities option is given, then the lines are drawn with the
    given intensity. The intensity is a number between 0 and 1.
  The function also admits the option \"Tooltips\". If this option is
     set to True, then the lines will have a tooltip that shows the
    wavelength of the line. If this option is set to False, then the
    lines will not have a tooltip. The default value for this option
    is True.
  If \"Tooltips\" is set to True and the option \"Comments\" is a non
    -empty list, then the tooltip will append the wavelength and the
    values in the comments list for the tooltips.
  The function also admits the option \"SpectrumFunction\". This
    option is a function that takes a wavelength and returns a color.
    The default value for this option is WaveToRGB.
  ";
  SpectrumPlot[lines_, widthToHeightAspect_, lineWidth_, opts :
    OptionsPattern[]] := Module[
    {minWave, maxWave, height, fuzzyLines},
  (
    colorFun = OptionValue["SpectrumFunction"];
    {minWave, maxWave} = MinMax[lines];
    height     = (maxWave - minWave)/widthToHeightAspect;
    fuzzyLines = Which[
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] == 0,
        FuzzyRectangle[#, lineWidth, 0, height, colorFun[#]] &/@
    lines,
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]]
    == 0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1]] &,
    {lines, lineWidth}],
      NumberQ[lineWidth] && Length[OptionValue["Intensities"]] > 0,
        MapThread[FuzzyRectangle[#1, lineWidth, 0, height, colorFun
    [#1], #2] &, {lines, OptionValue["Intensities"]}],
      Not[NumberQ[lineWidth]] && Length[OptionValue["Intensities"]] >
     0,
        MapThread[FuzzyRectangle[#1, #2, 0, height, colorFun[#1], #3]
     &, {lines, lineWidth, OptionValue["Intensities"]}]
    ];
    comments = Which[
      Length[OptionValue["Comments"]] > 0,
        MapThread[StringJoin[ToString[#1]<>" nm","\n",ToString[#2]]&,
        {lines, OptionValue["Comments"]}],
      Length[OptionValue["Comments"]] == 0,
        ToString[#] <>" nm" & /@ lines,
      True,
      {}
    ];
    If[OptionValue["Tooltips"],
      fuzzyLines = MapThread[Tooltip[#1, #2] &, {fuzzyLines, comments
    }];
    ];
    graphicsOpts = FilterRules[{opts}, Options[Graphics]];
    Graphics[fuzzyLines,
        graphicsOpts,
        Background -> Black,
        PlotRange -> {{minWave, maxWave}, {0, height}}]
    )
  ];

End[];

EndPackage[];
```

12.5 misc.m

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

```
BeginPackage["misc`"];

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

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

FirstOrderPerturbation;
SecondOrderPerturbation;
RoundValueWithUncertainty;

ToPythonSymPyExpression;
RoundToSignificantFigures;
RobustMissingQ;

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

Begin["`Private`"];

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

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

  BlockMatrixMultiply::usage="BlockMatrixMultiply[A,B] gives the
    matrix multiplication of A and B, with A and B having a compatible
     block structure that allows for matrix multiplication into a
    congruent block structure.";
  BlockMatrixMultiply[Amat_,Bmat_]:=Module[{rowIdx,colIdx,sumIdx},
  (
    Table[
      Sum[Amat[[rowIdx,sumIdx]].Bmat[[sumIdx,colIdx]],{sumIdx,1,
    Dimensions[Amat][[2]]}],
    {rowIdx,1,Dimensions[Amat][[1]]},
    {colIdx,1,Dimensions[Bmat][[2]]}
    ]
  )
  ];

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

```
    n>=0]][[1]];
    blockSubIndex = Mod[index-accumulatedBlockSize[[blockIndex]],
   blockSizes[[blockIndex]],1];
    Return[{blockIndex,blockSubIndex}]
  )
  ];

  TruncateBlockArray::usage="TruncateBlockArray[blockArray,
    truncationIndices, blockWidths] takes a an array of blocks and
    selects the columns and rows corresponding to truncationIndices.
    The indices being given in what would be the ArrayFlatten[
    blockArray] version of the array. They blocks in the given array
    may be SparseArray. This is equivalent to FlattenArray[blockArray
    ][truncationIndices, truncationIndices] but may be more efficient
    blockArray is sparse.";
  TruncateBlockArray[blockArray_,truncationIndices_,blockWidths_]:=
    Module[
  {truncatedArray,blockCol,blockRow,blockSubCol,blockSubRow},(
  truncatedArray = Table[
    {blockCol,blockSubCol} = BlockAndIndex[blockWidths,fullColIndex];
    {blockRow,blockSubRow} = BlockAndIndex[blockWidths,fullRowIndex];
    blockArray[[blockRow,blockCol]][[blockSubRow,blockSubCol]],
  {fullColIndex,truncationIndices},
  {fullRowIndex,truncationIndices}
  ];
  Return[truncatedArray]
  )
  ];

  BlockArrayDimensionsArray::usage="BlockArrayDimensionsArray[
    blockArray] returns the array of block sizes in a given blocked
    array.";
  BlockArrayDimensionsArray[blockArray_]:=(
    Map[Dimensions,blockArray,{2}]
  );

  ArrayBlocker::usage="ArrayBlocker[anArray, blockSizes] takes a flat
     2d array and a congruent 2D array of block sizes, and with them
    it returns the original array with the block structure imposed by
    blockSizes. The resulting array satisfies ArrayFlatten[
    blockedArray] == anArray, and also Map[Dimensions, blockedArray
    ,{2}] == blockSizes.";
  ArrayBlocker[anArray_,blockSizes_]:=Module[{rowStart,colStart,
    colEnd,numBlocks,blockedArray,blockSize,rowEnd,aBlock,idxRow,
    idxCol},(
    rowStart   = 1;
    colStart   = 1;
    colEnd     = 1;
    numBlocks  = Length[blockSizes];
    blockedArray = Table[(
      blockSize  = blockSizes[[idxRow,idxCol]];
      rowEnd     = rowStart+blockSize[[1]]-1;
      colEnd     = colStart+blockSize[[2]]-1;
      aBlock     = anArray[[rowStart;;rowEnd,colStart;;colEnd]];
      colStart   = colEnd+1;
      If[idxCol==numBlocks,
        rowStart=rowEnd+1;
        colStart=1;
      ];
      aBlock
    ),
    {idxRow,1,numBlocks},
    {idxCol,1,numBlocks}
    ];
    Return[blockedArray]
  )
  ];

  ReplaceDiagonal::usage =
    "ReplaceDiagonal[matrix, repValue] replaces all the diagonal of
    the given array to the given value. The array is assumed to be
    square and the replacement value is assumed to be a number. The
    returned value is the array with the diagonal replaced. This
    function is useful for setting the diagonal of an array to a given
     value. The original array is not modified. The given array may be
     sparse.";
```

```
ReplaceDiagonal[matrix_, repValue_] :=
  ReplacePart[matrix,
    Table[{i, i} -> repValue, {i, 1, Length[matrix]}]];

Options[RoundValueWithUncertainty] = {"SetPrecision" -> False};
RoundValueWithUncertainty::usage =
  "RoundValueWithUncertainty[x,dx] given a number x together with an \
uncertainty dx this function rounds x to the first significant figure \
of dx and also rounds dx to have a single significant figure.
The returned value is a list with the form {roundedX, roundedDx}.
The option \"SetPrecision\" can be used to control whether the \
Mathematica precision of x and dx is also set accordingly to these \
rules, otherwise the rounded numbers still have the original \
precision of the input values.
If the position of the first significant figure of x is after the \
position of the first significant figure of dx, the function returns \
{0,dx} with dx rounded to one significant figure.";
RoundValueWithUncertainty[x_, dx_, OptionsPattern[]] := Module[
  {xExpo, dxExpo, sigFigs, roundedX, roundedDx, returning},
  (
    xExpo = RealDigits[x][[2]];
    dxExpo = RealDigits[dx][[2]];
    sigFigs = xExpo - dxExpo + 1;
    {roundedX, roundedDx} = If[sigFigs <= 0,
      {0., N@RoundToSignificantFigures[dx, 1]},
      N[
      {
        RoundToSignificantFigures[x, xExpo - dxExpo + 1],
        RoundToSignificantFigures[dx, 1]}
      ]
      ];
    returning = If[
      OptionValue["SetPrecision"],
      {SetPrecision[roundedX, Max[1, sigFigs]],
      SetPrecision[roundedDx, 1]},
      {roundedX, roundedDx}
      ];
    Return[returning]
    )
  ];

RoundToSignificantFigures::usage =
  "RoundToSignificantFigures[x, sigFigs] rounds x so that it only has \
sigFigs significant figures.";
RoundToSignificantFigures[x_, sigFigs_] :=
  Sign[x]*N[FromDigits[RealDigits[x, 10, sigFigs]]];

RobustMissingQ[expr_] := (FreeQ[expr, _Missing] === False);

TextBasedProgressBar[progress_, totalIterations_, prefix_:""] :=
 Module[
    {progMessage},
    progMessage = ToString[progress] <> "/" <> ToString[
 totalIterations];
    If[progress < totalIterations,
        WriteString["stdout", StringJoin[prefix, progMessage, "\r"
 ]],
        WriteString["stdout", StringJoin[prefix, progMessage, "\n"
 ]]
    ];
];

FirstOrderPerturbation::usage="Given the eigenVectors of a matrix A
   (which doesn't need to be given) together with a corresponding
  perturbation matrix perMatrix, this function calculates the first
  derivative of the eigenvalues with respect to the scale factor of
  the perturbation matrix. In the sense that the eigenvalues of the
  matrix A + β perMatrix are to first order equal to \[Lambda] + \[
  Delta]_i β, where the \[Delta]_i are the returned values. This
  assuming that the eigenvalues are non-degenerate.";
FirstOrderPerturbation[eigenVectors_,
```

```
174     perMatrix_] := (Diagonal[
175     eigenVectors . perMatrix . Transpose[eigenVectors]])
176
177   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.";
178   SecondOrderPerturbation[eigenValues_, eigenVectors_, perMatrix_] :=
        (
179     dim = Length[perMatrix];
180     eigenBras = Conjugate[eigenVectors];
181     eigenKets = eigenVectors;
182     matV = Abs[eigenBras . perMatrix . Transpose[eigenKets]]^2;
183     OneOver[x_, y_] := If[x == y, 0, 1/(x - y)];
184     eigenDiffs = Outer[OneOver, eigenValues, eigenValues, 1];
185     pProduct = Transpose[eigenDiffs]*matV;
186     Return[2*(Total /@ Transpose[pProduct])];
187     )
188
189   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.";
190   SuperIdentity[args___] := {args};
191
192   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.";
193   FlattenBasis[basis_] := Module[{flatbasis, flatbasisLS,
        flatbasisNums},
194     (
195       flatbasis = Flatten[basis];
196       flatbasisLS = flatbasis[[1 ;; ;; 4]];
197       flatbasisNums = Select[flatbasis, Not[StringQ[#]] &];
198       Return[{flatbasisLS, flatbasisNums}]
199       )
200     ];
201
202   RecoverBasis::usage="RecoverBasis[{flatBasisLS, flatbasisNums}]
        takes the output of FlattenBasis and returns the original basis.
        The input is a list with two elements {flatbasisLS, flatbasisNums
        }.";
203   RecoverBasis[{flatbasisLS_, flatbasisNums_}] := Module[{recBasis},
204     (
205     recBasis = {{{#[[1]], #[[2]]}, #[[3]]}, #[[4]]} & /@ (Flatten /@
206         Transpose[{flatbasisLS,
207           Partition[Round[2*#]/2 & /@ flatbasisNums, 3]}]);
208     Return[recBasis];
209     )
210     ]
211
212   ExtractSymbolNames[expr_Hold] := Module[
213     {strSymbols},
214     strSymbols = ToString[expr, InputForm];
215     StringCases[strSymbols, RegularExpression["\\w+"]][[2 ;;]]
216     ]
217
218   ExportToH5::usage =
219     "ExportToH5[fname, Hold[{symbol1, symbol2, ...}]] takes an .h5
        filename and a held list of symbols and export to the .h5 file the
         values of the symbols with keys equal the symbol names. The
        values of the symbols cannot be arbitrary, for instance a list
        with mixes numbers and string will fail, but an Association with
        mixed values exports ok. Do give it a try.
220     If the file is already present in disk, this function will
        overwrite it by default. If the value of a given symbol contains
```

```
    symbolic numbers, e.g. \[Pi], these will be converted to floats in
     the exported file.";
221 Options[ExportToH5] = {"Overwrite" -> True};
222 ExportToH5[fname_String, symbols_Hold, OptionsPattern[]] := (
223   If[And[FileExistsQ[fname], OptionValue["Overwrite"]],
224   (
225     Print["File already exists, overwriting ..."];
226     DeleteFile[fname];
227     )
228   ];
229   symbolNames = ExtractSymbolNames[symbols];
230   Do[(Print[symbolName];
231     Export[fname, ToExpression[symbolName], {"Datasets", symbolName
    },
232     OverwriteTarget -> "Append"]
233     ), {symbolName, symbolNames}]
234   )
235
236 GreedyMatching::usage="GreedyMatching[aList, bList] returns a list
    of pairs of elements from aList and bList that are closest to each
     other, this is returned in a list together with a mapping of
    indices from the aList to those in bList to which they were
    matched. The option \"alistLabels\" can be used to specify labels
    for the elements in aList. The option \"blistLabels\" can be used
    to specify labels for the elements in bList. If these options are
    used, the function returns a list with three elements the pairs of
     matched elements, the pairs of corresponding matched labels, and
    the mapping of indices.";
237 Options[GreedyMatching] = {
238     "alistLabels" -> {},
239     "blistLabels" -> {}};
240 GreedyMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
241   aValues = aValues0,
242   bValues = bValues0,
243   bValuesOriginal = bValues0,
244   bestLabels, bestMatches,
245   bestLabel, aElement, givenLabels,
246   aLabels, aLabel,
247   diffs, minDiff,
248   bLabels,
249   minDiffPosition, bestMatch},
250   (
251   aLabels     = OptionValue["alistLabels"];
252   bLabels     = OptionValue["blistLabels"];
253   bestMatches = {};
254   bestLabels  = {};
255   givenLabels = (Length[aLabels] > 0);
256   Do[
257     (
258     aElement        = aValues[[idx]];
259     diffs           = Abs[bValues - aElement];
260     minDiff         = Min[diffs];
261     minDiffPosition = Position[diffs, minDiff][[1, 1]];
262     bestMatch       = bValues[[minDiffPosition]];
263     bestMatches     = Append[bestMatches, {aElement, bestMatch}];
264     If[givenLabels,
265       (
266       aLabel     = aLabels[[idx]];
267       bestLabel  = bLabels[[minDiffPosition]];
268       bestLabels = Append[bestLabels, {aLabel, bestLabel}];
269       bLabels    = Drop[bLabels, {minDiffPosition}];
270       )
271       ];
272     bValues = Drop[bValues, {minDiffPosition}];
273     If[Length[bValues] == 0, Break[]];
274     ),
275     {idx, 1, Length[aValues]}
276     ];
277   pairedIndices = MapIndexed[{#2[[1]], Position[bValuesOriginal,
    #1[[2]]][[1, 1]]} &, bestMatches];
278   If[givenLabels,
279     Return[{bestMatches, bestLabels, pairedIndices}],
280     Return[{bestMatches, pairedIndices}]
281     ]
282   )
283   ]
```

```
284
285   StochasticMatching::usage="StochasticMatching[aValues, bValues]
        finds a better assignment by randomly shuffling the elements of
        aValues and then applying the greedy assignment algorithm. The
        function prints what is the range of total absolute differences
        found during shuffling, the standard deviation of all of them, and
         the number of shuffles that were attempted. The option \"
        alistLabels\" can be used to specify labels for the elements in
        aValues. The option \"blistLabels\" can be used to specify labels
        for the elements in bValues. If these options are used, the
        function returns a list with three elements the pairs of matched
        elements, the pairs of corresponding matched labels, and the
        mapping of indices.";
286   Options[StochasticMatching] = {"alistLabels" -> {},
287     "blistLabels" -> {}};
288   StochasticMatching[aValues0_, bValues0_, numShuffles_ : 200,
        OptionsPattern[]] := Module[{
289     aValues = aValues0,
290     bValues = bValues0,
291     matchingLabels, ranger, matches, noShuff, bestMatch, highestCost,
         lowestCost, dev, sorter, bestValues,
292     pairedIndices, bestLabels, matchedIndices, shuffler
293     },
294     (
295     matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
296     ranger = Range[1, Length[aValues]];
297     matches = If[Not[matchingLabels], (
298         Table[(
299           shuffler = If[i == 1, ranger, RandomSample[ranger]];
300           {bestValues, matchedIndices} =
301           GreedyMatching[aValues[[shuffler]], bValues];
302           cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
303           {cost, {bestValues, matchedIndices}}
304           ), {i, 1, numShuffles}]
305         ),
306       Table[(
307         shuffler = If[i == 1, ranger, RandomSample[ranger]];
308         {bestValues, bestLabels, matchedIndices} =
309           GreedyMatching[aValues[[shuffler]], bValues,
310           "alistLabels" -> OptionValue["alistLabels"][[shuffler]],
311           "blistLabels" -> OptionValue["blistLabels"]];
312         cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
313         {cost, {bestValues, bestLabels, matchedIndices}}
314         ), {i, 1, numShuffles}]
315       ];
316     noShuff = matches[[1, 1]];
317     matches = SortBy[matches, First];
318     bestMatch = matches[[1, 2]];
319     highestCost = matches[[-1, 1]];
320     lowestCost = matches[[1, 1]];
321     dev = StandardDeviation[First /@ matches];
322     Print[lowestCost, " <-> ", highestCost, " | \[Sigma]=", dev,
323       " | N=", numShuffles, " | null=", noShuff];
324     If[matchingLabels,
325       (
326       {bestValues, bestLabels, matchedIndices} = bestMatch;
327       sorter = Ordering[First /@ bestValues];
328       bestValues = bestValues[[sorter]];
329       bestLabels = bestLabels[[sorter]];
330       pairedIndices =
331         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
332         bestValues];
333       Return[{bestValues, bestLabels, pairedIndices}]
334       ),
335       (
336       {bestValues, matchedIndices} = bestMatch;
337       sorter = Ordering[First /@ bestValues];
338       bestValues = bestValues[[sorter]];
339       pairedIndices =
340         MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
341         bestValues];
342       Return[{bestValues, pairedIndices}]
343       )
344       ];
345     )
346     ]
```

```
347
348  FlowMatching::usage="FlowMatching[aList, bList] returns a list of
       pairs of elements from aList and bList that are closest to each
       other, this is returned in a list together with a mapping of
       indices from the aList to those in bList to which they were
       matched. The option \"alistLabels\" can be used to specify labels
       for the elements in aList. The option \"blistLabels\" can be used
       to specify labels for the elements in bList. If these options are
       used, the function returns a list with three elements the pairs of
        matched elements, the pairs of corresponding matched labels, and
       the mapping of indices. This is basically a wrapper around
       Mathematica's FindMinimumCostFlow function. By default the option
       \"notMatched\" is zero, and this means that all elements of aList
       must be matched to elements of bList. If this is not the case, the
        option \"notMatched\" can be used to specify how many elements of
        aList can be left unmatched. By default the cost function is Abs
       [#1-#2]&, but this can be changed with the option \"CostFun\",
       this function needs to take two arguments.";
349  Options[FlowMatching] = {"alistLabels" -> {}, "blistLabels" -> {},
       "notMatched" -> 0, "CostFun"-> (Abs[#1-#2] &)};
350  FlowMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
351    aValues = aValues0, bValues = bValues0, edgesSourceToA,
       capacitySourceToA, nA, nB,
352    costSourceToA, midLayer, midLayerEdges, midCapacities,
353    midCosts, edgesBtoSink, capacityBtoSink, costBtoSink,
354    allCapacities, allCosts, allEdges, graph,
355    flow, bestValues, bestLabels, cFun,
356    aLabels, bLabels, pairedIndices, matchingLabels},
357    (
358    matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
359    aLabels = OptionValue["alistLabels"];
360    bLabels = OptionValue["blistLabels"];
361    cFun    = OptionValue["CostFun"];
362    nA      = Length[aValues];
363    nB      = Length[bValues];
364    (*Build up the edges costs and capacities*)
365    (*From source to the nodes representing the values of the first \
366  list*)
367    edgesSourceToA    = ("source" \[DirectedEdge] {"A", #}) & /@
       Range[1, nA];
368    capacitySourceToA = ConstantArray[1, nA];
369    costSourceToA     = ConstantArray[0, nA];
370
371    (*From all the elements of A to all the elements of B*)
372    midLayer = Table[{{"A", i} \[DirectedEdge] ({"B", j}), 1, cFun[
       aValues[[i]], bValues[[j]]]}, {i, 1, nA}, {j, 1, nB}];
373    midLayer = Flatten[midLayer, 1];
374    {midLayerEdges, midCapacities, midCosts} = Transpose[midLayer];
375
376    (*From the elements of B to the sink*)
377    edgesBtoSink    = ({"B", #} \[DirectedEdge] "sink") & /@ Range[1,
        nB];
378    capacityBtoSink = ConstantArray[1, nB];
379    costBtoSink     = ConstantArray[0, nB];
380
381    (*Put it all together*)
382    allCapacities = Join[capacitySourceToA, midCapacities,
       capacityBtoSink];
383    allCosts      = Join[costSourceToA, midCosts, costBtoSink];
384    allEdges      = Join[edgesSourceToA, midLayerEdges, edgesBtoSink
       ];
385    graph         = Graph[allEdges, EdgeCapacity -> allCapacities,
386      EdgeCost -> allCosts];
387
388    (*Solve it*)
389    flow          = FindMinimumCostFlow[graph, "source", "sink", nA -
        OptionValue["notMatched"], "OptimumFlowData"];
390    (*Collect the pairs of matched indices*)
391    pairedIndices = Select[flow["EdgeList"], And[Not[#[[1]] === "
       source"], Not[#[[2]] === "sink"]] &];
392    pairedIndices = {#[[1, 2]], #[[2, 2]]} & /@ pairedIndices;
393    (*Collect the pairs of matched values*)
394    bestValues    = {aValues[[#[[1]]]], bValues[[#[[2]]]]} & /@
       pairedIndices;
395    (*Account for having been given labels*)
396    If[matchingLabels,
```

```
      (
      bestLabels = {aLabels[[#[[1]]]], bLabels[[#[[2]]]]} & /@
  pairedIndices;
      Return[{bestValues, bestLabels, pairedIndices}]
      ),
      (
      Return[{bestValues, pairedIndices}]
      )
      ];
    )
    ]

  HelperNotebook::usage="HelperNotebook[nbName] creates a separate
    notebook and returns a function that can be used to print to the
    bottom of it. The name of the notebook, nbName, is optional and
    defaults to OUT.";
  HelperNotebook[nbName_:"OUT"] :=
  Module[{screenDims, screenWidth, screenHeight, nbWidth, leftMargin,
    PrintToOutputNb}, (
    screenDims =
      SystemInformation["Devices", "ScreenInformation"][[1, 2, 2]];
    screenWidth = screenDims[[1, 2]];
    screenHeight = screenDims[[2, 2]];
    nbWidth = Round[screenWidth/3];
    leftMargin = screenWidth - nbWidth;
    outputNb = CreateDocument[{}, WindowTitle -> nbName,
      WindowMargins -> {{leftMargin, Automatic}, {Automatic,
          Automatic}}, WindowSize -> {nbWidth, screenHeight}];
    PrintToOutputNb[text_] :=
      (
          SelectionMove[outputNb, After, Notebook];
          NotebookWrite[outputNb, Cell[BoxData[ToBoxes[text]], "
  Output"]];
      );
    Return[PrintToOutputNb]
    )
    ]

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

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

  (*Helper function to translate to Python/SymPy expressions*)
  ToPythonSymPyExpression::usage="ToPythonSymPyExpression[expr]
    converts a Mathematica expression to a SymPy expression. This is a
     little iffy and might break if the expression includes
    Mathematica functions that haven't been given a SymPy equivalent."
    ;
  ToPythonSymPyExpression[expr0_] := Module[{standardForm, expr=expr0
    },
    standardForm = StandardFormExpression[expr];
    StringReplace[standardForm, {
      "Power[" -> "Pow(",
      "Sqrt[" -> "sqrt(",
      "[" -> "(",
      "]" -> ")",
      "\\" -> "",
      "I" -> "1j",
      (*Remove special Mathematica backslashes*)
      "/" -> "/" (*Ensure division is represented with a slash*)}]];

  ToPythonSparseFunction[sparseArray_SparseArray, funName_] :=
    Module[{data, rowPointers, columnIndices, dimensions, pyCode,
    vars,
      varList, dataPyList,
      colIndicesPyList},(*Extract unique symbolic variables from the
    \
  SparseArray*)
    vars = Union[Cases[Normal[sparseArray], _Symbol, Infinity]];
    varList = StringRiffle[ToString /@ vars, ", "];
```

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

End[];
EndPackage[];
```

12.6 qalculations.m

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

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

workDir = DirectoryName[$InputFileName];

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

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

1. rmsDifference is the root mean squared difference between the
    calculated values and those quoted by Carnall;
2. carnallEnergies are the quoted calculated energies from Carnall
    used for comparison;
3. eigenEnergies are the calculated energies (in the case of an odd
    number of electrons the Kramers degeneracy may have been removed
    from this list according to the option \"Remove Kramers\");
4. ln is simply a string labelling the corresponding lanthanide;
5. carnallAssignments is a list of strings providing the multiplet
    assignments that Carnall assumed;
```

```
6. simplerStateLabels is a list of strings providing the multiplet assignments that this function assumes;
7. eigensys is a list of tuples where the first element is the energy corresponding to the eigenvector given as the second element (in the case of an odd number of electrons the Kramers degeneracy may have been removed from this list according to the option \"Remove Kramers\");
8. basis is a list that specifies the basis in which the Hamiltonian was constructed and diagonalized, equal to BasisLSJMJ[numE];
9. truncatedStates is the same as eigensys but with the truncated eigenvectors so that the total propability add up to at least eigenstateTruncationProbability.

This function admits the following options:
  - \"MakeNotebook\" -> True or False. If True, a notebook with a summary of the data is created. Default is True.
  - \"NotebookSave\" -> True or False. If True, the results notebook is saved automatically. Default is True.
  - \"eigenstateTruncationProbability\" -> 0.9. The probability sum of the truncated eigenvectors. Default is 0.9.
  - \"Include spin-spin\" -> True or False. If True, the spin-spin contribution to the magnetic interactions is included. Default is True.
  - \"Max Eigenstates in Table\" -> 100. The maximum number of eigenstates to be shown in the table shown in the results notebook. Default is 100.
  - \"Sparse\" -> True or False. If True, the numerical Hamiltonian is kept in sparse form. Default is True.
  - \"PrintFun\" -> Print, PrintTemporary, or other to serve as a printer for progress messages. Default is Print.
  - \"SaveData\" -> True or False. If True, the resulting data is saved to disk. Default is True.
  - \"ParamOverride\". An association that can override parameters in the Hamiltonian. Default is <||>. This override cannot change the inclusion or exclusion of the spin-spin contribution to the magnetic interactions, for this purpose use the option \"Include spin-spin\".
  - \"Append to Filename\" -> \"\". A string to append to the filename of the saved notebook and data files. Default is \"\".
  - \"Remove Kramers\" -> True or False. If True, the Kramers degeneracy is removed from the eigenstates. Default is True.
  - \"OutputDirectory\" -> \"calcs\". The directory where the output files are saved. Default is \"calcs\".
  - \"Explorer\" -> True or False. If True, the energy level diagram is interactive. Default is False.
";
Options[FastIonSolverLaF3] = {
  "MakeNotebook"      -> True,
  "NotebookSave"      -> True,
  "Include spin-spin" -> True,
  "eigenstateTruncationProbability" -> 0.9,
  "Max Eigenstates in Table" -> 100,
  "Sparse" -> True,
  "PrintFun" -> Print,
  "SaveData" -> True,
  "ParamOverride" -> <||>,
  "Append to Filename" -> "",
  "Remove Kramers" -> True,
  "OutputDirectory" -> "calcs",
  "Explorer" -> False
};
FastIonSolverLaF3[numE_, OptionsPattern[]] := Module[
  {makeNotebook, eigenstateTruncationProbability, host,
  ln, terms, termNames, carnallEnergies, eigenEnergies,
   simplerStateLabels,
  eigensys, basis, assignmentMatches, stateLabels, carnallAssignments
   },
  (
    PrintFun = OptionValue["PrintFun"];
    makeNotebook = OptionValue["MakeNotebook"];
    eigenstateTruncationProbability = OptionValue["
   eigenstateTruncationProbability"];
    maxStatesInTable = OptionValue["Max Eigenstates in Table"];
    Duplicator[aList_] := Flatten[{#, #} & /@ aList];
    host = "LaF3";
    ParamOverride = OptionValue["ParamOverride"];
```

```
    ln = theLanthanides[[numE]];
    terms = AllowedNKSLJTerms[Min[numE, 14 - numE]];
    termNames = First /@ terms;
    (* For labeling the states, the degeneracy in some of the terms is elided *)
    PrintFun["> Calculating simpler term labels ..."];
    termSimplifier = Table[termN -> If[StringLength[termN] == 3,
      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;
    ];

    (* Apply the parameter override *)
    Do[params[key] = ParamOverride[key],
      {key, Keys[ParamOverride]}
    ];

    (* Import the symbolic Hamiltonian *)
    PrintFun["> Loading the symbolic Hamiltonian for this configuration ..."];
    startTime = Now;
    numH = 14 - numE;
    numEH = Min[numE, numH];
    C2vsimplifier = {
      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,
      T11p -> 0, T11 -> 0, T12 -> 0, T14 -> 0, T15 -> 0, T16 -> 0, T18 -> 0, T17 -> 0, T19 -> 0};
    (* If the necessary symbolicHamiltonian is define load if not make it *)
    simpleHam = If[
      ValueQ[symbolicHamiltonians[numEH]],
      symbolicHamiltonians[numEH],
      SimplerSymbolicHamMatrix[numE, C2vsimplifier, "PrependToFilename" -> "C2v-", "Overwrite" -> False]
    ];
    endTime  = Now;
    loadTime = QuantityMagnitude[endTime - startTime, "Seconds"];
    PrintFun[">> Loading the symbolic Hamiltonian took ", loadTime, " seconds."];

    (*Enforce the override to the spin-spin contribution to the magnetic interactions*)
    params[\[Sigma]SS] = If[OptionValue["Include spin-spin"], 1, 0];

    (*Everything that is not given is set to zero*)
    params = ParamPad[params, "Print" -> False];
    PrintFun[params];
    numHam = ReplaceInSparseArray[simpleHam, params];
    If[Not[OptionValue["Sparse"]],
        numHam = Normal[numHam]
    ];
    PrintFun["> Calculating the SLJ basis ..."];
    basis = BasisLSJMJ[numE];

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

```
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];
179     If[And[OddQ[numE],OptionValue["Remove Kramers"]],
180       parsedStates = parsedStates[[;; ;; 2]]
181     ];
182
183     stateLabels = #[[-1]] & /@ parsedStates;
184     simplerStateLabels = ((#[[2]] /. termSimplifier) <> ToString
        [#[[3]], InputForm]) & /@ parsedStates;
185
186     PrintFun[">> Truncating eigenvectors to given probability ..."];
187     startTime = Now;
188     truncatedStates = ParseStatesByProbabilitySum[eigensys, basis,
189       eigenstateTruncationProbability,
190       0.01];
191     endTime = Now;
192     truncationTime = QuantityMagnitude[endTime - startTime, "Seconds"
        ];
193     PrintFun[">>> Truncation took ", truncationTime, " seconds."];
194
```

```
195     If[makeNotebook,
196     (
197       PrintFun["> Putting together results in a notebook ..."];
198       energyDiagram = Framed[
199         EnergyLevelDiagram[eigensys, "Title" -> title,
200         "Explorer" -> OptionValue["Explorer"],
201         "Background" -> White]
202         , Background -> White, FrameMargins -> 50];
203       appToFname = OptionValue["Append to Filename"];
204       PrintFun[">> Comparing the term assignments between qlanth and
    Carnall ..."];
205       AssignmentMatchFunc = Which[
206         StringContainsQ[#[[1]], #[[2]]],
207         "\[Checkmark]",
208         True,
209         "X"] &;
210       assignmentMatches = AssignmentMatchFunc /@ Transpose[{
    carnallAssignments, simplerStateLabels[[;; Length[
    carnallAssignments]]]}];
211       assignmentMatches = {{"\[Checkmark]",
212         Count[assignmentMatches, "\[Checkmark]"]}, {"X",
213         Count[assignmentMatches, "X"]}};
214       labelComparison = (AssignmentMatchFunc /@ Transpose[{
    carnallAssignments, simplerStateLabels[[;; Length[
    carnallAssignments]]]}]);
215       labelComparison = PadRight[labelComparison, Length[
    simplerStateLabels], "-"];
216
217       statesTable = Grid[Prepend[{Round[#[[1]]], #[[2]]} & /@
218           truncatedStates[[;;Min[Length[eigensys],maxStatesInTable
    ]]], {"Energy/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
219           "\[Psi]"}], Frame -> All, Spacings -> {2, 2},
220           FrameStyle -> Blue,
221           Dividers -> {{False, True, False}, {True, True}}];
222       DefaultIfMissing[expr_]:= If[FreeQ[expr, Missing[]], expr,"NA"
    ];
223       PrintFun[">> Rounding the energy differences for table
    presentation ..."];
224       roundedDiffs = Round[diffs, 0.1];
225       roundedDiffs = PadRight[roundedDiffs, Length[simplerStateLabels
    ], "-"];
226       roundedDiffs = DefaultIfMissing /@ roundedDiffs;
227       diffs = PadRight[diffs, Length[simplerStateLabels], "-"];
228       diffs = DefaultIfMissing /@ diffs;
229       diffTableData = Transpose[{simplerStateLabels, eigenEnergies,
230         labelComparison,
231         PadRight[carnallAssignments, Length[simplerStateLabels], "-"
    ],
232         DefaultIfMissing/@PadRight[carnallEnergies, Length[
    simplerStateLabels], "-"],
233         roundedDiffs}
234       ];
235       diffTable = TableForm[diffTableData,
236         TableHeadings -> {None, {"qlanth",
237         "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)", "", "Carnall",
238         "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
239         "\[CapitalDelta]E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)"}}
240       ];
241
242       diffs = Sort[eigenEnergies][[;; Length[carnallEnergies]]] -
    carnallEnergies;
243       notBad = FreeQ[#,Missing[]]&/@diffs;
244       diffs = Pick[diffs, notBad];
245       (* diffHistogram = Histogram[diffs,
246         Frame -> True,
247         ImageSize -> 800,
248         AspectRatio -> 1/3, FrameStyle -> Directive[16],
249         FrameLabel -> {"(qlanth-carnall)/Ky", "Freq"}
250       ]; *)
251
252       rmsDifference = Sqrt[Total[diffs^2/Length[diffs]]];
253       labelTempate  = StringTemplate["\!\(\*SuperscriptBox[\(`ln`\),
    \(\(3\)\(+\)\)]\)"];
254       diffData   = diffs;
255       diffLabels = simplerStateLabels[[;;Length[notBad]]];
256       diffLabels = Pick[diffLabels, notBad];
```

```
257       diffPlot   = Framed[
258           ListLabelPlot[
259             diffData[[;;;;If[OddQ[numE],2,1]]],
260             diffLabels[[;;;;If[OddQ[numE],2,1]]],
261             Frame -> True,
262             PlotRange -> All,
263             ImageSize -> 1200,
264             AspectRatio -> 1/3,
265             Filling -> Axis,
266             FrameLabel -> {"",
267             "(qlanth-carnall) / \!\(\*SuperscriptBox[\(cm\), \(-1\)
    ]\)"},
268             PlotMarkers -> "OpenMarkers",
269             PlotLabel ->
270             Style[labelTempate[<|"ln" -> ln|>] <> " | " <> "\[Sigma]=
    " <>
271               ToString[Round[rmsDifference, 0.01]] <>
272               " \!\(\*SuperscriptBox[\(cm\), \(-1\)]\)\n", 20],
273             Background -> White
274           ],
275           Background -> White,
276           FrameMargins -> 50
277         ];
278       (* now place all of this in a new notebook *)
279       nb = CreateDocument[
280       {
281         TextCell[Style[
282           DisplayForm[RowBox[{SuperscriptBox[host <> ":" <> ln, "3+"
    ], "(", SuperscriptBox["f", numE], ")"}]]
283           ], "Title", TextAlignment -> Center
284         ],
285         TextCell["Energy Diagram",
286           "Section",
287           TextAlignment -> Center
288         ],
289         TextCell[energyDiagram,
290           TextAlignment -> Center
291         ],
292         TextCell["Multiplet Assignments & Energy Levels",
293           "Section",
294           TextAlignment -> Center
295         ],
296         (* TextCell[diffHistogram, TextAlignment -> Center], *)
297         TextCell[diffPlot, "Output", TextAlignment -> Center],
298         TextCell[assignmentMatches, "Output", TextAlignment -> Center
    ],
299         TextCell[diffTable, "Output", TextAlignment -> Center],
300         TextCell["Truncated Eigenstates", "Section", TextAlignment ->
     Center],
301         TextCell["These are some of the resultant eigenstates which
    add up to at least a total probability of " <> ToString[
    eigenstateTruncationProbability] <> ".", "Text", TextAlignment ->
    Center],
302         TextCell[statesTable, "Output", TextAlignment -> Center]
303       },
304       WindowSelected -> True,
305       WindowTitle -> ln <> " in " <> "LaF3" <> appToFname,
306       WindowSize -> {1600, 800}];
307       If[OptionValue["SaveData"],
308       (
309         exportFname = FileNameJoin[{workDir,OptionValue["
    OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".mx"
    }];
310         SelectionMove[nb, After, Notebook];
311         NotebookWrite[nb, Cell["Reload Data", "Section",
    TextAlignment -> Center]];
312         NotebookWrite[nb,
313           Cell[(
314             "{rmsDifference, carnallEnergies, eigenEnergies, ln,
    carnallAssignments, simplerStateLabels, eigensys, basis,
    truncatedStates} = Import[FileNameJoin[{NotebookDirectory[],\"" <>
     StringSplit[exportFname,"/"][[-1]] <> "\"}]];"
315           ),"Input"
316         ]
317       ];
318       NotebookWrite[nb,
```

```
319           Cell[(
320             "Manipulate[First[MinimalBy[truncatedStates, Abs[First[#]
     - energy] &]], {energy,0}]"
321             ),"Input"]
322         ];
323         (* Move the cursor to the top of the notebook *)
324         SelectionMove[nb, Before, Notebook];
325         Export[exportFname,
326           {rmsDifference, carnallEnergies, eigenEnergies, ln,
    carnallAssignments, simplerStateLabels, eigensys, basis,
    truncatedStates}
327         ];
328         tinyexportFname = FileNameJoin[
329           {workDir, OptionValue["OutputDirectory"], ln <> " in " <> "
    LaF3" <> appToFname <> " - tiny.m"}
330         ];
331         tinyExport = <|"ln"->ln,
332                        "carnallEnergies"->carnallEnergies,
333                        "rmsDifference"-> rmsDifference,
334                        "eigenEnergies"-> eigenEnergies,
335                        "carnallAssignments"-> carnallAssignments,
336                        "simplerStateLabels" -> simplerStateLabels|>;
337         Export[tinyexportFname, tinyExport];
338       )
339       ];
340       If[OptionValue["NotebookSave"],
341         (
342           nbFname = FileNameJoin[{workDir,OptionValue["
    OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".nb"
    }];
343           PrintFun[">> Saving notebook to ", nbFname, " ..."];
344           NotebookSave[nb, nbFname];
345         )
346       ];
347     )
348     ];
349
350     Return[{rmsDifference, carnallEnergies,
351             eigenEnergies, ln,
352             carnallAssignments, simplerStateLabels,
353             eigensys, basis,
354             truncatedStates}];
355   )
356 ];
357
358 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:
359     \[Psi]i:simple, (* main contribution to the wavefuction |i>*)
360     \[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]];
403   {rmsDifference,carnallEnergies,eigenEnergies,ln,
404   carnallAssignments,simplerStateLabels,eigensys,basis,
        truncatedStates} = Import["./examples/"<>ln<>" in LaF3 - example.
        mx"];
405
406   (* Some of the above are not needed here *)
407   Clear[truncatedStates];
408   Clear[basis];
409   Clear[rmsDifference];
410   Clear[carnallEnergies];
411   Clear[carnallAssignments];
412   If[OddQ[numE],
413       eigenEnergies      = eigenEnergies[[;;;;2]];
414       simplerStateLabels = simplerStateLabels[[;;;;2]];
415       eigensys           = eigensys[[;;;;2]];
416   ];
417   eigenEnergies = eigenEnergies - eigenEnergies[[1]];
418
419   magIon = <||>;
420   PrintFun["Calculating the magnetic dipole line strength array..."];
421   magIon["Line Strength"] = magIon;
422   MagDipLineStrength[eigensys, numE, "Reload MagOp" -> False, "Units"
        -> "SI"];
423
424   PrintFun["Calculating the M1 spontaneous transition rates ..."];
425   magIon["AMD"] = MagDipoleRates[eigensys, numE, "Units"->"SI","
        Lifetime"->False];
426   magIon["AMD"] = magIon["AMD"]/.{0.->Indeterminate};
427
428   PrintFun["Calculating the oscillator strength for transition from
        the ground state ..."];
429   magIon["fMD"]=GroundStateOscillatorStrength[eigensys, numE];
430
431   PrintFun["Calculating the natural radiative lifetims ..."];
432   magIon["Radiative lifetimes"]= 1/magIon["AMD"];
433
434   PrintFun["Calculating the transition energies in K ..."];
435   transitionEnergies=Outer[Subtract,First/@eigensys,First/@eigensys];
436   magIon["Transition Energies / K"]=ReplaceDiagonal[
        transitionEnergies,Indeterminate];
437
438   PrintFun["Calculating the transition wavelengths in nm ..."];
439   magIon["Transition Wavelengths in nm"] =10^7/magIon["Transition
        Energies / K"];
440
```

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

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

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

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

(* tabular output *)
basename     = ln <> " in " <> host <> " - example - " <> "MD1 - tabular.zip";
exportFname  = FileNameJoin[{"./examples",basename}];
PrintFun["Exporting tabular data to "<>exportFname<>" ..."];
exportKey    = StringReplace[basename,".zip"->".m"];
Export[exportFname, <|exportKey->finalTable|>];

(* raw data output *)
basename    = ln <> " in " <> host <> " - example - " <> "MD1 - raw.zip";
rawexportFname = FileNameJoin[{"./examples",basename}];
PrintFun["Exporting raw data as an association to "<>exportFname<>" ..."];
rawexportKey   = StringReplace[basename,".zip"->".m"];
Export[rawexportFname, <|rawexportKey->magIon|>];

(* csv output *)
PrintFun["Formatting and exporting a CSV output..."];
csvOut = Table[
    StringJoin[Riffle[ToString[#,CForm]&/@finalTable[[i]],","]],
{i,1,Length[finalTable]}
];
csvOut       = StringJoin[Riffle[csvOut, "\n"]];
basename     = ln <> " in " <> host <> " - example - " <> "MD1.csv";
exportFname = FileNameJoin[{"./examples", basename}];
PrintFun["Exporting csv data to "<>exportFname<>" ..."];
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

[BG15] Cristiano Benelli and Dante Gatteschi. *Introduction to molecular magnetism: from transition metals to lanthanides*. John Wiley & Sons, 2015.

[BG34] R. F. Bacher and S. Goudsmit. "Atomic Energy Relations. I". In: *Phys. Rev.* 46.11 (Dec. 1934). Publisher: American Physical Society, pp. 948–969. DOI: 10.1103/PhysRev.46.948. URL: https://link.aps.org/doi/10.1103/PhysRev.46.948.

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

[Car+89] W. T. Carnall et al. "A systematic analysis of the spectra of the lanthanides doped into single crystal LaF3". en. In: *The Journal of Chemical Physics* 90.7 (1989), pp. 3443–3457. ISSN: 0021-9606, 1089-7690. DOI: 10.1063/1.455853. URL: http://aip.scitation.org/doi/10.1063/1.455853 (visited on 07/02/2021).

[CFW65] W To Carnall, PR Fields, and BG Wybourne. "Spectral intensities of the trivalent lanthanides and actinides in solution. I. Pr3+, Nd3+, Er3+, Tm3+, and Yb3+". In: *The Journal of Chemical Physics* 42.11 (1965). Publisher: American Institute of Physics, pp. 3797–3806.

[Che+08] Xueyuan Chen et al. "A few mistakes in widely used data files for fn configurations calculations". In: *Journal of luminescence* 128.3 (2008). Publisher: Elsevier, pp. 421–427.

[Cow81] Robert Duane Cowan. *The theory of atomic structure and spectra*. en. Los Alamos series in basic and applied sciences 3. Berkeley: University of California Press, 1981. ISBN: 978-0-520-03821-9.

[DR06] Chang-Kui Duan and Michael F Reid. "Dependence of the spontaneous emission rates of emitters on the refractive index of the surrounding media". In: *Journal of alloys and compounds* 418.1-2 (2006). Publisher: Elsevier, pp. 213–216.

[DZ12] Christopher M. Dodson and Rashid Zia. "Magnetic dipole and electric quadrupole transitions in the trivalent lanthanide series: Calculated emission rates and oscillator strengths". en. In: *Physical Review B* 86.12 (Sept. 2012), p. 125102. ISSN: 1098-0121, 1550-235X. DOI: 10.1103/PhysRevB.86.125102. URL: https://link.aps.org/doi/10.1103/PhysRevB.86.125102 (visited on 07/02/2021).

[JCC68] BR Judd, HM Crosswhite, and Hannah Crosswhite. "Intra-atomic magnetic interactions for f electrons". In: *Physical Review* 169.1 (1968). Publisher: APS, p. 130. DOI: https://doi.org/10.1103/PhysRev.169.130.

[JS84] BR Judd and MA Suskin. "Complete set of orthogonal scalar operators for the configuration f^3". In: *JOSA B* 1.2 (1984). Publisher: Optica Publishing Group, pp. 261–265. DOI: https://doi.org/10.1364/JOSAB.1.000261.

[Jud62] B. R. Judd. "Optical Absorption Intensities of Rare-Earth Ions". en. In: *Physical Review* 127.3 (Aug. 1962), pp. 750–761. ISSN: 0031-899X. DOI: 10.1103/PhysRev.127.750. URL: https://link.aps.org/doi/10.1103/PhysRev.127.750 (visited on 07/02/2021).

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

[Lin74] Ingvar Lindgren. "The Rayleigh-Schrodinger perturbation and the linked-diagram theorem for a multi-configurational model space". In: *Journal of Physics B: Atomic and Molecular Physics* 7.18 (1974). Publisher: IOP Publishing, p. 2441.

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

[Ofe62] GS Ofelt. "Intensities of crystal spectra of rare-earth ions". In: *The journal of chemical physics* 37.3 (1962). Publisher: American Institute of Physics, pp. 511–520.

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

[Rud07] Zenonas Rudzikas. *Theoretical atomic spectroscopy*. 2007.

[RW63] K Rajnak and BG Wybourne. "Configuration interaction effects in l^N configurations". In: *Physical Review* 132.1 (1963). Publisher: APS, p. 280. DOI: https://doi.org/10.1103/PhysRev.132.280.

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

[Tre52] R. E. Trees. "The L (L + 1) Correction to the Slater Formulas for the Energy Levels". en. In: *Physical Review* 85.2 (Jan. 1952), pp. 382–382. ISSN: 0031-899X. DOI: 10.1103/PhysRev.85.382. URL: https://link.aps.org/doi/10.1103/PhysRev.85.382 (visited on 01/18/2022).

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

[Wyb63] BG Wybourne. “Electrostatic Interactions in Complex Electron Configurations”. In: *Journal of Mathematical Physics* 4.3 (1963). Publisher: American Institute of Physics, pp. 354–356.

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

Index

configuration interaction, 16

forced electric diple transitions, 48

Judd-Ofelt theory, 48

Kayser, 45

Laporte's rule, 48
level, 1

semi-empirical approach, 16
spherical harmonics, 39
state, 1

term, 1