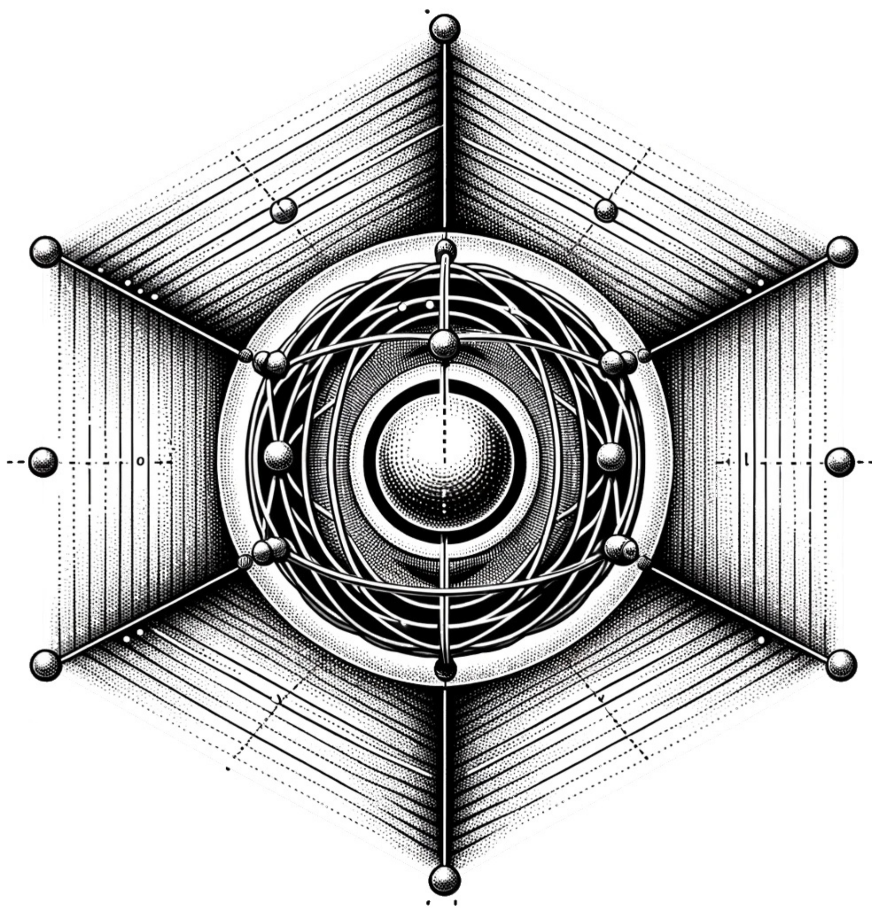

`qlanth`

Version 1.0

Juan David Lizarazo Ferro

February 17, 2024

qlanth

`qlanth` is a Mathematica package that can be used to calculate the level structure of lanthanide ions embedded in crystals. For this purpose it uses a single configuration description with an effective Hamiltonian described below. This Hamiltonian aims to describe the observed properties of ions embedded in solids in a picture that imagines them as free-ions but modified by the influence of the lattice in which they find themselves in.

This picture is one that developed and mostly matured in the second half of the last century from the efforts of Brian Judd, Hannah Crosswhite, Michael Reid, Bill Carnall, Brian Wybourne, Katherine Rajnak, and others. The goal of this code is to provide a modern implementation of the calculations that resulted from their work, with the aim of fixing some small errors that might have been included at the time these calculations were made. It also aims to provide useful electronic versions of the data these Hamiltonians may produce, including energies and eigenvectors.

`qlanth` also includes data that might be of use to those interested in the single-configuration description of lanthanide ions, separate to their specific use in this code. These data include the coefficients of fractional parentage (as calculated by Velkov and parsed here), and reduced matrix elements for all the operators listed above in the effective Hamiltonian. These are provided as standard Mathematica associations that should be simple to use elsewhere.

The included Mathematica notebook `qlanth.nb` has examples of the capabilities that this package offers, and the `/examples` folder includes a series of notebooks for most of the trivalent lanthanide ions in lanthanum fluoride. LaF3 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. In them, the fact that the parameters for different ions vary in regular fashion, provides some validity to this effective Hamiltonian as a physically reasonable description.

This code was originally authored by Christopher Dodson and Rashid Zia and has been modified and rewritten by David Lizarazo. It has also benefited from conversations with Tharnier Puel at the University of Iowa.

1 The effective Hamiltonian

Electrons in a multi-electron ion are subject to several interactions. Firstly, they are attracted to the nucleus around which they orbit. Additionally, they experience repulsion from other electrons. Electrons also possess spin, subjecting them 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. Finally, among pairs of electrons, the spin of one can influence the other through the interaction of their respective magnetic dipoles.

This framework sufficiently describes the interactions within a free ion. However, to extend this model to ions within a crystal, we must incorporate the effects of the crystal field. This is often achieved by considering the electric field that an ion experiences from the surrounding charges in the crystal lattice, a concept referred to as the crystal field effect.

The Hilbert space of a multi-electron ion is a large auditorium. In principle the Hilbert space should have a countable infinity of discrete states and a 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 effectively be described as an aggregate of an inert core and a fewer active valence electrons.

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

Here we shall assume a single-configuration description. Meaning that all the valence electrons in the ions that we study here will all be considered to be located in f-orbitals, or what is the same, that they are described by $\underline{f}^n$ wavefunctions. This is, however, a harsh approximation, but 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 we call *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. What results from this is are operator that now act solely within the single configuration but with a convoluted coefficient that depends on overlaps between different configurations. This coefficient one could try to evaluate, and there are some that have trodden this road. Others simply label that complex expression with an unassuming symbol, and leave it as a parameter that one can hope to fit against experimental data. It is from this that the parameters $\alpha, \beta, \gamma, P^0, P^2,$ and P^4 enter into the description that we shall use here.

Something that is also borne out of the configuration interaction analysis is that their influence also modifies previously present intra-configuration operators. For instance, part of the configuration interaction influence that results from the Coulomb repulsion between electrons brings about new operators that need to be included, but they also contribute to the intra-configuration Slater integrals. As such, every parameter in the Hamiltonian becomes a quantity to be fitted against spectroscopic data.

When finding the matrix elements of the Hamiltonian defined by these terms, one also requires the specification of the basis in which the matrix elements will be computed. What we shall use here are states determined by five quantum numbers: the total orbital angular momentum L, the total spin angular momentum S, the total angular momentum J, and the projection of the total angular momentum along the z-axis M_J. To account for the fact that there might be a few different ways to amount for a given LS, it becomes necessary to have a fifth quantum number that discriminates between these different cases. This other quantum number we shall simply call α, which in the notation of Nielson and Koster is simply an integer number that enumerates all the possible LS in a given $\underline{\mathrm{f}}^n$configuration.

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

$$\hat{\mathcal{H}} = \underbrace{\hat{\mathcal{H}}_{\mathrm{k}}}_{\text{kinetic}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:sn}}}_{\text{e:shielded nuc}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{e:e}}}_{\text{e:e}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:o}}}_{\text{spin-orbit}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:s}}}_{\substack{\text{spin:spin}\\ \text{and spin:other-orbit}}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{s:oo\oplus ecs:o}}}_{\substack{\text{spin:other-orbit}\\ \text{ec-correlated-spin:orbit}}} + \tag{1}$$

$$\underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(3)}}_{\text{Trees effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{G}_2}}_{\mathrm{G}_2\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\mathcal{SO}(7)}}_{\mathcal{SO}(7)\text{ effective op}} + \underbrace{\hat{\mathcal{H}}_{\curlywedge}}_{\substack{\text{effective}\\ \text{three-body}}} + \underbrace{\hat{\mathcal{H}}_{\mathrm{cf}}}_{\text{crystal field}} \tag{2}$$

$$\hat{\mathcal{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m_e}\sum_{i=1}^{n}\nabla_i^2 \text{ (kinetic energy of } n \text{ v-electrons)} \tag{3}$$

$$\hat{\mathcal{H}}_{\mathrm{e:sn}} = \sum_{i=1}^{n} V_{\mathrm{sn}}(\hat{r}_i) \text{ (interaction of v-electrons with shielded nuclear charge)} \tag{4}$$

$$\hat{\mathcal{H}}_{\mathrm{e:e}} = \sum_{i>j}^{n,n} \frac{e^2}{\|\hat{\vec{r}}_i - \hat{\vec{r}}_j\|} = \sum_{k=0,2,4,6} F^k \hat{f}_k \text{ (v-electron:v-electron repulsion)} \tag{5}$$

$$\hat{\mathcal{H}}_{\mathrm{s:o}} = \begin{cases} \sum_{i=1}^{n} \xi(r_i)\left(\hat{\underline{s}}_i \cdot \hat{\underline{\ell}}_i\right) \text{ with } \xi(r_i) = \frac{\hbar^2}{2m^2c^2 r_i}\frac{\mathrm{d}V_{\mathrm{sn}}(r_i)}{\mathrm{d}r_i} \\ \sum_{i=1}^{n} \zeta\left(\hat{\underline{s}}_i \cdot \hat{\underline{\ell}}_i\right) \substack{\text{with } \zeta \text{ the radial average of } \xi(r_i)\\ \text{or used as phenomenological parameter}} \end{cases} \tag{6}$$

$$\hat{\mathcal{H}}_{\mathrm{s:s}} = \sum_{k=0,2,4} M^{(k)} \hat{m}_k^{ss} \tag{7}$$

$$\hat{\mathcal{H}}_{\mathrm{s:oo\oplus ecs:o}} = \sum_{k=2,4,6} P^{(k)}\hat{p}_k + \sum_{k=0,2,4} M^{(k)}\hat{m}_k \tag{8}$$

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

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

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

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

$$\hat{\mathcal{H}}_{\curlywedge} = T'^{(2)} t_2' + \sum_{k=2,3,4,6,7,8}^{n} T^{(k)}\hat{t}_k \text{ (effective 3-body operators } \hat{t}_k) \tag{12}$$

$$\hat{\mathcal{H}}_{\mathrm{cf}} = \sum_{i=1}^{n} V_{\mathrm{CF}}(\hat{r}_i) = \sum_{i=1}^{n}\sum_{k=2,4,6}\sum_{q=-k}^{k} B_q^{(k)} \mathcal{C}_q^{(k)}(i) \substack{\text{(crystal field interaction of v-electrons with}\\ \text{electrostatic field due to surroundings)}} \tag{13}$$

$$\tag{14}$$

It is of some importance to note that the eigenstates that we'll end up with have shoved under the rug all the radial dependence of the wavefunctions. This dependence has been already integrated in the parameters that the Hamiltonian has.

1.1 $\hat{\mathscr{H}}_{\mathrm{k}}$: kinetic energy

$$\hat{\mathscr{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m_e}\sum_{i=1}^{N}\nabla_i^2 \text{ (kinetic energy of N v-electrons)} \tag{15}$$

Within the basis that we'll use, the kinetic energy simply contributes a constant energy shift, and since all we care about are energy transitions, then this term can be omitted from the analysis.

1.2 $\hat{\mathscr{H}}_{\mathrm{e:sn}}$: e:shielded nuc

$$\hat{\mathscr{H}}_{\mathrm{k}} = -\frac{\hbar^2}{2m_e}\sum_{i=1}^{N}\nabla_i^2 \text{ (kinetic energy of N v-electrons)} \tag{16}$$

Instead of using the shielded nuclear charge this could have been instead the bare nuclear charge, but then we would have needed to take into account the repulsion from the electrons in closed shells. Here we are already bringing some simplification in that we approximate the compound effect on the valence electrons due to the charge of the filled shells and the charge of the nucleus is that of a central field.

Then again, this term also contributes a common energy shift to all the energies that we can obtain within the single-configuration description, so this one will also be omittted. It might be useful to use this term and the previous one to estimate the energy differences between the states in different configurations, but we will not do that here.

1.3 $\hat{\mathscr{H}}_{\mathrm{e:e}}$: e:e repulsion

$$\hat{\mathscr{H}}_{\mathrm{e:e}} = \sum_{i>j}^{n,n}\frac{e^2}{\|\hat{\vec{r}}_i - \hat{\vec{r}}_j\|} = \sum_{k=0,2,4,6} F^k \hat{f}_k = \sum_{k=0,1,2,3} E_k \hat{e}^k \tag{17}$$

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

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

$$\langle \underline{\mathrm{f}}^n \alpha^{2S+1}L \| \hat{\mathscr{H}}_{\mathrm{e:e}} \| \underline{\mathrm{f}}^n \alpha'^{2S'+1}L' \rangle = \sum_{k=0,2,4,6} F^k f_k(n, \alpha LS, \alpha' L'S') \tag{18}$$

where

$$f_k(n,\alpha LS,\alpha' L'S') = \frac{1}{2}\delta(S,S')\delta(L,L')\langle \underline{\mathrm{f}}\|\hat{C}^{(k)}\|\underline{\mathrm{f}}\rangle^2 \times \\ \left\{ \frac{1}{2L+1}\sum_{\alpha'' L''}\langle \underline{\mathrm{f}}^n\alpha'' L'' S\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha LS\rangle\langle \underline{\mathrm{f}}^n\alpha'' L'' S\|\hat{U}^{(k)}\|\underline{\mathrm{f}}^n \alpha' LS\rangle - \delta(\alpha,\alpha')\frac{n\,(4\underline{\mathrm{f}}+2-n)}{(2\underline{\mathrm{f}}+1)(4\underline{\mathrm{f}}+1)} \right\} \tag{19}$$

1.4 $\hat{\mathscr{H}}_{\mathrm{s:o}}$: The spin-orbit interaction

Here one can be of two minds, one can either start from a relativistic description, only including the interaction of charged particles. Then when descending to the non-relativistic description one will notice a term that involves both the orbital angular momentum and the spin angular momentum. In this view the spin-orbit term arises as a relativistic correction to the non-relativistic Schrodinger equation.

From the non-relativistic viewpoint one may also take it as a given that the electron has an associated magnetic moment. From this one would then continue to consider the effect that magnetic fields have on it. One of those fields that one due to the motion of the electron around the nucleus, one would then conclude a term that involves both the spin and the orbital motion of the electron.

More generally one may picture an electron in a radial electrostatic potential $V(r)$, in which case the energy associated to the spin-orbit is

$$\hat{h}_{\mathrm{s:o}} = \frac{\hbar^2}{2\mathrm{m}_e^2 c^2}\left(\frac{1}{r}\frac{\mathrm{d}V}{\mathrm{d}r}\right)\hat{l}\cdot\hat{s} := \zeta(r)\hat{l}\cdot\hat{s}. \tag{20}$$

And adding up for all the n valence electrons

$$\hat{\mathscr{H}}_{\text{s:o}} = \sum_{i}^{n} \zeta(r_i)\hat{l}_i\cdot\hat{s}_i. \tag{21}$$

The matrix elements that we then require are

$$\begin{aligned}\langle\alpha LSJM_J|\hat{\mathscr{H}}_{\text{s:o}}|\alpha' L'S'J'M_{J'}\rangle &= \zeta\delta(J,J')\delta(M_J,M_{J'})\langle\alpha LSJM_J|\sum_{i}^{n}\hat{l}_i\cdot\hat{s}_i|\alpha' L'S'JM_J\rangle \\ &= \zeta(-1)^{J+L+S'}\begin{Bmatrix} L & L' & 1 \\ S' & S & J\end{Bmatrix}\langle\alpha LS|\sum_{i}^{n}\hat{l}_i\cdot\hat{s}_i|\alpha' L'S'\rangle \\ &= \zeta(-1)^{J+L+S'}\begin{Bmatrix} L & L' & 1 \\ S' & S & J\end{Bmatrix}\sqrt{\underline{\ell}(\underline{\ell}+1)(2\underline{\ell}+1)}\langle\alpha LS\|\hat{V}^{(11)}\|\alpha' L'S'\rangle \end{aligned} \tag{22}$$

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

$$\hat{V}^{(11)} = \sum_{i=1}^{n}\left(\hat{s}\hat{u}^{(1)}\right)_i, \tag{23}$$

where the rank on the spin operator $\hat{s}$ has been omitted, and the rank of the tensor operator shown 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 (1965):

$$\begin{aligned}\langle\underline{\ell}^n\psi\|\hat{V}^{(1k)}\|\underline{\ell}^n\psi'\rangle = \langle\underline{\ell}^n\alpha LS\|\hat{V}^{(1k)}\|\underline{\ell}^n\alpha' L'S'\rangle &= n\sqrt{\underline{s}(\underline{s}+1)(2\underline{s}+1)}\sqrt{[\![S]\!]\,[\![L]\!]\,[\![S']\!]\,[\![L']\!]}\times \\ \sum_{\bar{\psi}}(-1)^{\bar{S}+\bar{L}+S+L+\underline{\ell}+\underline{s}+k+1}\left(\psi\{\!|\bar{\psi}\right)\left(\bar{\psi}|\!\}\psi'\right)&\begin{Bmatrix} S & S' & 1 \\ \underline{s} & \underline{s} & \bar{S}\end{Bmatrix}\begin{Bmatrix} L & L' & k \\ \underline{\ell} & \underline{\ell} & \bar{L}\end{Bmatrix}\end{aligned} \tag{24}$$

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

We also calculate $V^{(1k)}$ since they are useful for calculating the matrix elements of XXXXX.

1.5 $\hat{\mathscr{H}}_{\text{cf}}$: A crystal-field

The picture of an ion inside of a crystal is lacking in at least two respects. First, we are imagining that the ion and the lattice can be neatly separated, in that the electrons in the ion are not shared with bonds to the surrounding lattice. Second we

Within this view we would like to add in some manner the influence of the surrounding lattice. The simplest way of doing this considers the lattice as a static aggregate of charges. For this aggregate of charges we could associate an electrostatic potential descibed as a multipolar sum of the form:

$$V(r_i,\theta_i,\phi_i) = \sum_{k=1}^{\infty}\mathscr{A}_q^{(k)}r^k\mathcal{C}_q^{(k)}(\theta_i,\phi_i) \tag{25}$$

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

$$\hat{\mathscr{H}}_{\text{cf}}(\vec{r}) = \sum_{i=1}^{n}\sum_{k=0}^{\infty}\sum_{q=-k}^{(k)}\mathscr{A}_q^{(k)}r_i^k\mathcal{C}_q^{(k)}(\theta_i,\phi_i). \tag{26}$$

And if we average the radial coordinate,

$$\hat{\mathscr{H}}_{\text{cf}} = \sum_{i=1}^{n}\sum_{k=1}^{\infty}\sum_{q=-k}^{k}B_q^{(k)}\mathcal{C}_q^{(k)}(i) \tag{27}$$

where the radial average is included as

$$B_q^{(k)} := \mathscr{A}_q^{(k)}\langle r^k\rangle. \tag{28}$$

In principle the value for $B_q^{(k)}$ could have both real an imaginary parts, in qlanth this is taken into account by separating out the real and imaginary parts with the replacement in terms of two real-valued parameters

$$B_q^{(k)} \to B_q^{(k)} + iS_q^{(k)}. \tag{29}$$

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):

$$\langle \ell \| \hat{u}^{(k)} \| \ell' \rangle = \lhd(\ell, k, \ell'). \tag{30}$$

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

$$\hat{U}^{(k)} = \sum_{i}^{n} \hat{u}_i^{(k)}. \tag{31}$$

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

$$C_q^{(k)} = \langle \underline{\ell} \| C^{(k)} \| \underline{\ell}' \rangle \hat{u}_q^{(k)} = (-1)^{\underline{\ell}} \sqrt{[\![\underline{\ell}]\!]\,[\![\underline{\ell}']\!]} \begin{pmatrix} \underline{\ell} & k & \underline{\ell}' \\ 0 & 0 & 0 \end{pmatrix} \hat{u}_q^{(k)}. \tag{32}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{\mathscr{H}}_{\text{cf}} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-3}} = \sum_{k=1}^{\infty} \sum_{q=-k}^{k} B_q^{(k)} \langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' SL'J'M_{J'} \rangle \langle \underline{\ell} \| \hat{C}^{(k)} \| \underline{\ell} \rangle \tag{33}$$

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

$$\overbrace{\langle \underline{\ell}^n \alpha SLJM_J | \hat{U}_q^{(k)} | \underline{\ell}^n \alpha' S'L'J'M_{J'} \rangle}^{\text{Wybourne eqn. 6-4}} = (-1)^{J-M_J} \begin{pmatrix} J & k & J' \\ -M_J & q & M_{J'} \end{pmatrix} \langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle \tag{34}$$

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

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha SLJ \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Wybourne eqn. 6-5}} = (-1)^{S+L+J'+k} \sqrt{[\![J]\!]\,[\![J']\!]} \times \\ \begin{Bmatrix} J & J' & k \\ L' & L & S \end{Bmatrix} \langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle. \end{aligned} \tag{35}$$

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

$$\begin{aligned} \overbrace{\langle \underline{\ell}^n \alpha SL \| \hat{U}^{(k)} \| \underline{\ell}^n \alpha' S'L' \rangle}^{\text{Cowan eqn. 11.53}} = \delta(S, S') n (-1)^{\underline{\ell}+L+k} \sqrt{[\![L]\!]\,[\![L']\!]} \times \\ \sum_{\bar{\alpha}\bar{L}\bar{S}} (-1)^{\bar{L}} \begin{Bmatrix} \underline{\ell} & k & \underline{\ell} \\ L & \bar{L} & L' \end{Bmatrix} \left(\underline{\ell}^n \alpha LS \{\!| \underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} \right) \left(\underline{\ell}^{n-1} \bar{\alpha}\bar{L}\bar{S} |\!\} \underline{\ell}^n \alpha' L'S' \right). \end{aligned} \tag{36}$$

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

The above equations are implemented in `qlanth` by the function `CrystalField`. This function puts together the symbolic sum in eqn. (33) 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.

1.6 $\hat{\mathscr{H}}_{\mathscr{SO}(3)}, \hat{\mathscr{H}}_{\mathbf{G}_2}, \hat{\mathscr{H}}_{\mathscr{SO}(7)}$: Electrostatic configuration interaction

This is a first term where we take into account the very important contributions from configuration interaction. When the interaction with configurations ?? and ?? it was realized that the way the omission of these configurations in the single configuration description was to relax the previous restriction that F^k should only have even values for k. Parallel to this Trees noticed an interesting fact which is that a fair amount of correction to the calculated spectrum of would benefit if one added to all of the LS energies a term quadratic in L. Soon after this it was acknowledged that the inclusion of odd F^k was equivalent to adding three terms related to the Casimir operators of the groups $SO(3)$, G_2, and $SO(7)$. In addition to this, the configuration interaction analysis, also showed that the contributions from other configuration would also overlap with the already allowed even F^k.

Of these Casimir operators one of them is familiar to us as it is the Casimir operator of $SO(3)$, namely $\hat{L}^2$. In analogy to $\hat{L}^2$ in which the quantum number L can be used to determine the eigenvalues, in the cases of $\hat{\mathscr{H}}_{\mathrm{G}_2}$ the necessary state label is the U label of the LS term, and in the case of $\hat{\mathscr{H}}_{\mathscr{SO}(7)}$ the necessary label is W. If $\Lambda_{\mathrm{G}_2}(U)$ is used to note the eigenvalue of the Casimir operator of G_2 corresponding to label U, and $\Lambda_{\mathscr{SO}(7)}(W)$ the eigenvalue corresponding to state label W, then the matrix elements of $\hat{\mathscr{H}}_{\mathscr{SO}(3)}$, $\hat{\mathscr{H}}_{\mathrm{G}_2}$ and $\hat{\mathscr{H}}_{\mathscr{SO}(7)}$ are diagonal in all quantum numbers and are given by

$$\langle \underline{\ell}^n \alpha SLJM_J | \hat{\mathscr{H}}_{\mathscr{SO}(3)} | \underline{\ell}^n \alpha' S'L'J'M_J' \rangle = \alpha \delta(S,S')\delta(L,L')\delta(\alpha,\alpha')\delta(J,J')\delta(M_J,M_J')L(L+1) \tag{37}$$

$$\langle \underline{\ell}^n U \alpha SLJM_J | \hat{\mathscr{H}}_{\mathrm{G}_2} | \underline{\ell}^n U \alpha' S'L'J'M_J' \rangle = \beta \delta(S,S')\delta(L,L')\delta(\alpha,\alpha')\delta(J,J')\delta(M_J,M_J')\Lambda_{\mathrm{G}_2}(U) \tag{38}$$

$$\langle \underline{\ell}^n W \alpha SLJM_J | \hat{\mathscr{H}}_{\mathscr{SO}(7)} | \underline{\ell}^n W \alpha' S'L'J'M_J \rangle = \gamma \delta(S,S')\delta(L,L')\delta(\alpha,\alpha')\delta(J,J')\delta(M_J,M_J')\Lambda_{\mathscr{SO}(7)}(W) \tag{39}$$

In `qlanth` the role of $\Lambda_{\mathscr{SO}(7)}(W)$ is played by the function `GSO7W`, the role of $\Lambda_{\mathrm{G}_2}(U)$ by `GG2U`, and the role of $\Lambda_{\mathscr{SO}(3)}(L)$ 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.

1.7 $\hat{\mathscr{H}}_{\mathrm{s:s\text{-}s:oo}}$: Spin-spin and spin other orbit interaction

The calculation of the $\hat{\mathscr{H}}_{\mathrm{s:s\text{-}s:oo}}$ 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 need to take precomputed values for reduced matrix elements either in $\underline{\mathrm{f}}^2$ or in $\underline{\mathrm{f}}^3$ and then we "pull" the for all $\underline{\mathrm{f}}^n$ configuration with the help of the standar formulae involving coefficients of fractional parentage.

The analysis of spin-other-orbit, and the spin-spin contributions we use in `qlanth` is that of Judd, Crosswhite, and Crosswhite [JCC68]. If the spin-orbit correction arrived from the influence that the orbital motion of an electron has on its own magnetic moment, the spin-other-orbit reflects the interaction that the motion of one electron has on the magnetic moment of another. 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

$$\hat{\mathscr{H}}_B = -\frac{1}{2}e^2 \sum_{i>j} \left[(\alpha_i \cdot \alpha_j)\frac{1}{r_{ij}} + (\alpha_i \cdot \vec{r}_{ij})(\alpha_j \cdot \vec{r}_{ij})\frac{1}{r_{ij}^3} \right]. \tag{40}$$

When this relativistic equation is expanded in powers of v/c, a number of inter-electron interactions appear. Two of them being the spin-other-orbit and spin-spin interactions.

As usual the radial part of the Hamiltonian is averaged, which in this case gives appearance to the Marvin integrals

$$M^{(k)} := \frac{e^2\hbar^2}{8m^2c^2} \langle (nl)^2 | \frac{r_<^k}{r_>^{k+3}} | (nl)^2 \rangle \tag{41}$$

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

$$\hat{\mathscr{H}}_{s:s} = -2 \sum_{i\neq j} \sum_k M^{(k)} \sqrt{(k+1)(k+2)(2k+3)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle \langle \underline{\ell} \| C^{(k+2)} \| \underline{\ell} \rangle \left\{ \hat{w}_i^{(1,k)} \hat{w}_j^{(1,k+2)} \right\}^{(2,2)0} \tag{42}$$

and the one for spin-other-orbit

$$\begin{aligned} \hat{\mathscr{H}}_{s:oo} = \sum_{i\neq j} \sum_k & \sqrt{(k+1)(2\underline{\ell}+k+2)(2\underline{\ell}-k)} \times \\ & \left[\left\{ \hat{w}_i^{(0,k+1)} \hat{w}_j^{(1,k)} \right\}^{(11)0} \left\{ M^{(k-1)} \langle \underline{\ell} \| C^{(k+1)} \| \underline{\ell} \rangle^2 + 2M^{(k)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle^2 \right\} + \right. \\ & \left. \left\{ \hat{w}_i^{(0,k)} \hat{w}_j^{(1,k+1)} \right\}^{(11)0} \left\{ M^{(k)} \langle \underline{\ell} \| C^{(k)} \| \underline{\ell} \rangle^2 + 2M^{(k-1)} \langle \underline{\ell} \| C^{(k+1)} \| \underline{\ell} \rangle^2 \right\} \right]. \end{aligned} \tag{43}$$

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

$$\langle \underline{\ell} \| \hat{w}^{(\kappa,k)} \| \underline{\ell} \rangle = \sqrt{[\![\kappa]\!]\,[\![k]\!]} \triangleleft(l,\kappa,l) \triangleleft(l,k,l) \tag{44}$$

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

$$\sqrt{5}\hat{\mathscr{T}}_0^{(22)} := \hat{\mathscr{H}}_{\text{s:s}} \tag{45}$$

$$-\sqrt{3}\hat{\mathscr{T}}_0^{(11)} := \hat{\mathscr{H}}_{\text{s:oo}} \tag{46}$$

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

$$\langle \gamma SLJ|\hat{\mathscr{H}}|\gamma' S' L' J'\rangle = \delta(J,J') \begin{Bmatrix} S' & L' & J \\ L & S & t \end{Bmatrix} \langle \gamma SL \| \hat{\mathscr{T}}^{(tt)} \| \gamma' S' L'\rangle. \tag{47}$$

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

For two-electron operators such as these, the matrix elements in $\underline{f}^n$ are related to those in $\underline{f}^{n-1}$ via:

$$\langle \underline{f}^n \psi \| \hat{\mathscr{T}}^{(tt)} \| \psi' \underline{f}^n\rangle = \frac{n}{n-2} \sum_{\bar{\psi},\bar{\psi}'} (-1)^{\bar{S}+\bar{L}+\underline{s}+\underline{\ell}+S'+L'} \sqrt{[\![S]\!]\,[\![S']\!]\,[\![L]\!]\,[\![L']\!]} \times \left(\psi\{\!|\bar{\psi}\right)\left(\psi'\{\!|\bar{\psi}'\right) \begin{Bmatrix} S & t & S' \\ \bar{S}' & \underline{s} & \bar{S} \end{Bmatrix} \begin{Bmatrix} L & t & L' \\ \bar{L}' & \underline{\ell} & \bar{L} \end{Bmatrix} \tag{48}$$

Where the sum runs over the terms $\bar{\psi}$ and $\bar{\psi}'$ in $\underline{f}^{n-1}$ which are parents common to ψ and ψ'. Using these the matrix elements of $\hat{\mathscr{T}}^{(11)}$ and $\hat{\mathscr{T}}^{(22)}$ in $\underline{f}^2$ can be used to compute all the reduced matrix elements in $\underline{f}^n$ and then these can be use, together with eqn 47 to obtain the matrix elements of $\hat{\mathscr{H}}_{\text{s:s}}$ and $\hat{\mathscr{H}}_{\text{s:oo}}$.

These equations are implemented in **qlanth** through the following functions: `GenerateT22Table`, `GenerateT11Table`, `ReducedT22infn`, `ReducedT22inf2`, `ReducedT11inf2`. Where `ReducedT22inf2` and `ReducedT11inf2` provide the reduced matrix elements for $\hat{\mathscr{T}}^{(11)}$ and $\hat{\mathscr{T}}^{(22)}$ in $\underline{f}^2$ as provided in table II of [JCC68].

1.8 $\hat{\mathscr{H}}_{\text{ecs:o}}$: Electrostatically correlated spin orbit & a note about configuration interaction

In the same paper [JCC68] that describes the spin-spin and spin-other-orbits consideration is also given to the emergence of additional corrections due to configuration interaction as described by the following operator (page. 169 of [JCC68])

$$\hat{\mathscr{H}}_{\text{ci}} = -\sum_{\chi}\sum_{i} \frac{1}{E_\chi}\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right)|\chi\rangle\langle\chi|\hat{\mathfrak{C}} - \frac{1}{E_\chi}\hat{\mathfrak{C}}|\chi\rangle\langle\chi|\xi(r_i)\left(\hat{\underline{s}}_i\cdot\hat{\underline{\ell}}_i\right) \tag{49}$$

where $\xi(r_h)(\hat{\underline{s}}_h\cdot\hat{\underline{\ell}}_h)$ is the customary spin-orbit interaction, E_χ is the energy of state $|\chi\rangle$, i is a label for the valence electrons, and $|\chi\rangle$ are states in the configurations to which one is "interacting" with.

Most importantly in the above, the term $\hat{\mathfrak{C}}$ stands for the non-central part of the Coulomb interaction. This serves as a reminder that the central field approximation of single-electron wavefunctions is, indeed, an approximation. The non-central part of the electrostatic field is defined as what remains after subtracting the radial component. This term is crucial to keep in mind because it facilitates parity-breaking transitions, such as forced electric dipole transitions. Moreover, the non-central nature of this term plays a significant role in configuration mixing (see [MR71]), which is why the operator $\hat{\mathfrak{C}}$ is prominently featured in the expression for configuration interaction, and why the modifier "electrostatically correlated" is prepended to spin-orbit and form "electrostatically correlated spin orbit". It's also worth keeping in mind that the derivation of such an expression is based on second-order perturbation theory.

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

$$\hat{\mathscr{H}}_{\text{ci}} = -\sqrt{3}\,\hat{t}_0^{(11)} \tag{50}$$

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

$$P^{(k)} = 6\sum_{f'} \frac{\zeta_{ff'}}{E_{ff'}} R^{(k)}(ff,ff') \text{ for } k = 0,2,4,6. \tag{51}$$

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

$$\zeta_{ff'} := \langle f|\xi(r)|f'\rangle \tag{52}$$

$$R^{(k)}(ff,ff') := e^2\langle f_1 f_2|\frac{r_<^k}{r_>^{k+1}}|f_1 f_2'\rangle. \tag{53}$$

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.

Judd *et al.* also develop 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

$$\langle\psi\|\hat{\mathscr{T}}^{(11)}+\hat{t}^{(11)}\|\psi'\rangle=\sum a_i\langle\psi\|\hat{z}_i\|\psi'\rangle \tag{54}$$

At this point a subtle point needs to be taken into account. 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 the obtaining a parametric Hamiltonian that can be fit with uncorrelated parameters, it is then necessary to subtract this part from $\hat{\mathscr{T}}^{(11)}+\hat{t}^{(11)}$. 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:

$$\hat{\mathscr{H}}_{\text{s:oo}}+\hat{\mathscr{H}}_{\text{ecs:o}}=\hat{\mathscr{T}}^{(11)}+\hat{t}^{(11)}-\frac{1}{6}a_{13}\hat{z}_{13} \tag{55}$$

where

$$a_{13}=-33M^{(0)}+3M^{(2)}+15/11M^{(4)}-6P^{(0)}+3/2(35P^{(2)}+77P^{(4)}+143P^{(6)}) \tag{56}$$

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

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 $\hat{\mathscr{T}}^{(22)}$ which are stored in the association `T22Table`.

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 $\underline{\text{f}}^2$, and `ReducedT22infn` for configurations above $\underline{\text{f}}^2$. When `ReducedT22infn` is called the sum in eqn. 48 is carried out using $t=2$. When `ReducedT22inf2` is called the reduced matrix elements from [JCC68] are used.

The function `GenerateSOOandECSOTable` calls the function `SOOandECSO` over all possible combinations of the arguments $\{n, SL, S'L', J\}$ and uses their values to pupulate the association `SOOandECSOTable`. In turn the function `SOOandECSO` queries the precomputed values of eqn. 55 as stored in the association `SOOandECSOLSTable`.

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 $\underline{\text{f}}^2$, and `ReducedSOOandECSOinfn` for configurations above $\underline{\text{f}}^2$. When `ReducedSOOandECSOinfn` is called the sum in eqn. 48 is carried out using $t=1$. When `ReducedSOOandECSOinf2` is called the reduced matrix elements from [JCC68] are used.

2 Notation

$$\overbracket{m}^{\text{mass of the electron}} \tag{57}$$

$$\overbracket{\hat{l}}^{\text{orbital angular momentum operator of a single electron}} \tag{58}$$

$$\overbracket{\hat{L}}^{\text{total orbital angular momentum operator}} \tag{59}$$

$$\overbracket{\hat{s}}^{\text{spin angular momentum operator of a single electron}} \tag{60}$$

$$\overbracket{\hat{S}}^{\text{total spin angular momentum operator}} \tag{61}$$

$$\overbracket{\Lambda}^{\text{Shorthand for all other quantum numbers}} \tag{62}$$

$$\overbracket{\underline{\ell}}^{\text{orbital angular momentum number}} \tag{63}$$

$$\overbracket{\underline{s}}^{\text{spinning angular momentum number}} \tag{64}$$

$$\overbracket{\hat{\mathfrak{C}}}^{\text{Coulomb non-central potential}} \tag{65}$$

$$\overbracket{\langle \Lambda LS \| \hat{O} \| \Lambda' L' S' \rangle}^{\text{LS-reduced matrix element of operator } \hat{O} \text{ between } \Lambda LS \text{ and } \Lambda' L' S'} \tag{66}$$

$$\overbracket{\langle \Lambda LSJ \| \hat{O} \| \Lambda' L' S' J' \rangle}^{\text{LSJ-reduced matrix element of operator } \hat{O} \text{ between } \Lambda LSJ \text{ and } \Lambda' L' S' J'} \tag{67}$$

$$\overbracket{{}^{2S+1}\alpha L \equiv |\alpha LS\rangle}^{\text{Spectroscopic term } \alpha LS \text{ in Russel-Saunders notation}} \tag{68}$$

$$\overbracket{\hat{X}^{(k)}}^{\text{spherical tensor operator of rank k}} \tag{69}$$

$$\overbracket{\hat{X}^{(k)}_q}^{\text{q-component of the spherical tensor operator } \hat{X}^{(k)}} \tag{70}$$

$$\overbracket{\left(\underline{\ell}^{n-1}\alpha' L' S' |\!\} \underline{\ell}^{n}\alpha LS\right)}^{\text{The coefficient of fractional parentage from the parent term } |\underline{\ell}^{n-1}\alpha' L' S'\rangle \text{ for the daughter term } |\underline{\ell}^{n}\alpha LS\rangle} \tag{71}$$

3 Definitions

$$\overbracket{[\![x]\!] := 2x+1}^{\text{two plus one}} \tag{72}$$

$$\overbracket{\hat{u}^{(k)}}^{\text{irreducible unit tensor operator of rank k}} \tag{73}$$

$$\overbracket{\hat{U}^{(k)} := \sum_{i=1}^{n} \hat{u}^{(k)}}^{\text{symmetric unit tensor operator for n equivalent electrons}} \tag{74}$$

$$\overbracket{\left(\underline{\ell}^{n-1}\alpha' L' S' |\!\} \underline{\ell}^{n}\alpha LS\right)}^{\text{The coefficient of fractional parentage from the parent term } |\underline{\ell}^{n-1}\alpha' L' S'\rangle \text{ for the daughter term } |\underline{\ell}^{n}\alpha LS\rangle} \tag{75}$$

$$\overbracket{C^{(k)}_q := \sqrt{\frac{4\pi}{2k+1}}\mathrm{Y}^{(k)}_q}^{\text{Renormalized spherical harmonics}} \tag{76}$$

$$\overbracket{\triangleleft(j_1, j_2, j_3) := \begin{cases} 1 & \text{if } j_1 = (j_2+j_3), (j_2+j_3-1), \ldots, |j_2-j_3| \\ 0 & \text{otherwise} \end{cases}}^{\text{Triangle “delta” between } j_1, j_2, j_3} \tag{77}$$

4 qlanth.m

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

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

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

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

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

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

REFERENCES:

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

ζ: spin-orbit strength parameter.

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

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

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

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

P0: 0th parameter for the two-body electrostatically correlated spin-
    orbit interaction
```

```
P2: 2nd parameter for the two-body electrostatically correlated spin-
    orbit interaction
P4: 4th parameter for the two-body electrostatically correlated spin-
    orbit interaction
P6: 6th parameter for the two-body electrostatically correlated spin-
    orbit interaction

gs: electronic gyromagnetic ratio

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

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

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

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

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

S24: crystal field parameter S_2^4 (real)
S26: crystal field parameter S_2^6 (real)
S34: crystal field parameter S_3^4 (real)
S36: crystal field parameter S_3^6 (real)

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

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

Bx: x component of external magnetic field (in T)
By: y component of external magnetic field (in T)
Bz: z component of external magnetic field (in T)
";
paramSymbols = StringSplit[paramAtlas, "\n"];
paramSymbols = Select[paramSymbols, # != ""& ];
paramSymbols = ToExpression[StringSplit[#, ":"][[1]]] & /@
    paramSymbols;
Protect /@ paramSymbols;
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}
];

(* Parameter families*)
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};

AllowedJ;
AllowedMforJ;
AllowedNKSLJMforJMTerms;
AllowedNKSLJMforJTerms;

AllowedNKSLJTerms;
AllowedNKSLTerms;
AllowedNKSLforJTerms;
AllowedSLJMTerms;
AllowedSLJTerms;

AllowedSLTerms;
BasisLSJMJ;
Bqk;
CFP;
CFPAssoc;

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

Cqk;
CrystalField;
Dk;
ElectrostaticConfigInteraction;
Electrostatic;

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

FindNKLSTerm;
FindSL;

FtoE;
GG2U;
GSO7W;
GenerateCFP;
GenerateCFPAssoc;

GenerateCFPTable;
GenerateCrystalFieldTable;
GenerateElectrostaticTable;
GenerateReducedUkTable;
GenerateReducedV1kTable;

GenerateSOOandECSOLSTable;
GenerateSOOandECSOTable;
GenerateSpinOrbitTable;
GenerateSpinSpinTable;
GenerateT22Table;

GenerateThreeBodyTables;
GenerateThreeBodyTables;
```

```
Generator;
GroundStateOscillatorStrength;
HamMatrixAssembly;
HamiltonianForm;

HamiltonianMatrixPlot;
HoleElectronConjugation;
IonSolver;
ImportMZip;
JJBlockMatrix;
JJBlockMagDip;
JJBlockMatrixFileName;

JJBlockMatrixTable;
LabeledGrid;
LoadAll;
LoadCFP;
LoadCarnall;

LoadChenDeltas;
LoadElectrostatic;
LoadGuillotParameters;
LoadParameters;
LoadSOOandECSO;

LoadSOOandECSOLS;
LoadSpinOrbit;
LoadSpinSpin;
LoadSymbolicHamiltonians;
LoadT11;

LoadT22;
LoadTermLabels;
LoadThreeBody;
LoadUk;
LoadV1k;

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

ParamPad;
ParseStates;
ParseStatesByNumBasisVecs;
ParseStatesByProbabilitySum;
ParseTermLabels;

Phaser;
PrettySaunders;
PrettySaundersSLJ;
PrettySaundersSLJmJ;
PrintL;

PrintSLJ;
PrintSLJM;
ReducedSOOandECSOinf2;
ReducedSOOandECSOinfn;
ReducedT11inf2;

ReducedT22inf2;
ReducedUk;
ReducedUkTable;
ReducedV1kTable;
Reducedt11inf2;

ReplaceInSparseArray;
SimplerSymbolicHamMatrix;
SOOandECSO;
SOOandECSOTable;
Seniority;
```

```

ShiftedLevels;
SixJay;
SpinOrbit;
SpinSpin;
SpinSpinTable;

Sqk;
SquarePrimeToNormal;
ReducedT22infn;
TPO;

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

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

fnTermLabels;
moduleDir;
symbolicHamiltonians;

(* this selects the function that is applied
to calculated matrix elements *)
SimplifyFun = Expand;

Begin["`Private`"]

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

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

  TPO::usage="Two plus one.";
  TPO[args__] := Times @@ ((2*# + 1) & /@ {args});

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

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

  TriangleAndSumCondition::usage = "TriangleAndSumCondition[a, b, c]
    returns True if a, b, and c satisfy the triangle and sum
    conditions.";
  TriangleAndSumCondition[a_, b_, c_] := (And[Abs[b - c] <= a <= (b +
    c), IntegerQ[a + b + c]]);

  SquarePrimeToNormal::usage = "Given a list with the parts
    corresponding to the squared prime representation of a number,
    this function parses the result into standard notation.";
  SquarePrimeToNormal[squarePrime_] :=
  (
    radical = Product[Prime[idx1 - 1] ^ Part[squarePrime, idx1], {idx1
    , 2, Length[squarePrime]}];
    radical = radical /. {"A" -> 10, "B" -> 11, "C" -> 12, "D" -> 13};
    val = squarePrime[[1]] * Sqrt[radical];
    Return[val];
  );

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

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

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];
]

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

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

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

ReducedV1k::usage = "ReducedV1k[n, 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}]]]
    ];
  ReducedUkTable = Table[
    (
      counter = counter+1;
      {numE, 3, SL, SpLp, k} -> SimplifyFun[ReducedUk[numE, 3, SL,
 SpLp, k]]
    ),
    {numE, 1, numEmax},
    {SL,   AllowedNKSLTerms[numE]},
    {SpLp, AllowedNKSLTerms[numE]},
    {k, {0, 2, 4, 6}}
  ];
  ReducedUkTable = Association[Flatten[ReducedUkTable]];
  ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "
 ReducedUkTable.m"}];
  If[And[OptionValue["Progress"], frontEndAvailable],
    NotebookDelete[progBar]
  ];
  If[OptionValue["Export"],
    (
      Print["Exporting to file " <> ToString[ReducedUkTableFname]];
      Export[ReducedUkTableFname, ReducedUkTable];
    )
  ];
  Return[ReducedUkTable];
)

GenerateReducedV1kTable::usage = "GenerateReducedV1kTable[nmax,
 export calculates values for Vk1 and returns an association where
 the keys are lists of the form {n, SL, SpLp, 1}. If the option \"
 Export\" is set to True then the resulting data is saved to ./data
 /ReducedV1kTable.m."
Options[GenerateReducedV1kTable] = {"Export" -> True, "Progress" ->
 True};
GenerateReducedV1kTable[numEmax_Integer:7, OptionsPattern[]]:= (
  numValues = Total[Length[AllowedNKSLTerms[#]]*Length[
 AllowedNKSLTerms[#]]&/@Range[1, numEmax]];
  Print["Calculating " <> ToString[numValues] <> " values for Vk1."
 ];
  counter = 1;
  If[And[OptionValue["Progress"], frontEndAvailable],
  progBar = PrintTemporary[
      Dynamic[Row[{ProgressIndicator[counter, {0, numValues}], " ",
        counter}]]]
    ];
  ReducedV1kTable = Table[
    (
      counter = counter+1;
      {n, SL, SpLp, 1} -> SimplifyFun[ReducedV1k[n, SL, SpLp, 1]]
    ),
    {n, 1, numEmax},
    {SL, AllowedNKSLTerms[n]},
    {SpLp, AllowedNKSLTerms[n]}
  ];
  ReducedV1kTable = Association[ReducedV1kTable];
  If[And[OptionValue["Progress"], frontEndAvailable],
    NotebookDelete[progBar]
  ];
  exportFname = FileNameJoin[{moduleDir, "data", "ReducedV1kTable.m"
 }];
  If[OptionValue["Export"],
    (
      Print["Exporting to file "<>ToString[exportFname]];
      Export[exportFname, ReducedV1kTable];
    )
  ];
  Return[ReducedV1kTable];
)

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

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

fsubk::usage = "Slater integral f_k. 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];
  ]

  fsupk::usage = "Super-script Slater integral f^k = Subscript[f, k] *
     Subscript[D, k]";
  fsupk[numE_, orbital_, NKSL_, NKSLp_ ,k_]:= (Dk[k] * fsubk[numE,
    orbital, NKSL, NKSLp, k])

  Dk::usage = "Ratio between the super-script and sub-scripted Slater
    integrals (F^k /F_k). k must be even. See table 6-3 in TASS, and
    also section 2-7 of Wybourne (1965). See also equation 6.41 in
    TASS.";
  Dk[k_] := {1, 225, 1089, 184041/25}[[k/2+1]]

  FtoE::usage = "FtoE[F0, F2, F4, F6] calculates the corresponding {E0
    , E1, E2, E3} values.
  See eqn. 2-80 in Wybourne. Note that in that equation the
    subscripted Slater integrals are used but since this function
    assumes the the input values are superscripted Slater integrals,
    it is necessary to convert them using Dk.";
  FtoE[F0_, F2_, F4_, F6_] := (Module[
    {E0, E1, E2, E3},
    E0 = (F0 - 10*F2/Dk[2] - 33*F4/Dk[4] - 286*F6/Dk[6]);
    E1 = (70*F2/Dk[2] + 231*F4/Dk[4] + 2002*F6/Dk[6])/9;
    E2 = (F2/Dk[2] - 3*F4/Dk[4] + 7*F6/Dk[6])/9;
    E3 = (5*F2/Dk[2] + 6*F4/Dk[4] - 91*F6/Dk[6])/3;
    Return[{E0, E1, E2, E3}];
  ]
  );

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

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

  GenerateElectrostaticTable::usage = "GenerateElectrostaticTable[
    numEmax] can be used to generate the table for the electrostatic
    interaction from f^1 to f^numEmax. If the option \"Export\" is set
     to True then the resulting data is saved to ./data/
    ElectrostaticTable.m.";
  Options[GenerateElectrostaticTable] = {"Export" -> True, "
    Coefficients" -> "Slater"};
  GenerateElectrostaticTable[numEmax_Integer:7, OptionsPattern[]]:= (
    ElectrostaticTable = Table[
      {numE, SL, SpLp} -> SimplifyFun[Electrostatic[{numE, SL, SpLp},
    "Coefficients" -> OptionValue["Coefficients"]]],
      {numE, 1, numEmax},
      {SL, AllowedNKSLTerms[numE]},
      {SpLp, AllowedNKSLTerms[numE]}
    ];
    ElectrostaticTable = Association[Flatten[ElectrostaticTable]];
    If[OptionValue["Export"],
      Export[FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"
    }],
      ElectrostaticTable];
    ];
    Return[ElectrostaticTable];
  )

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

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

  BasisTableGenerator::usage = "BasisTableGenerator[numE] 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}.";
BasisLSJMJ[numE_] := Module[{energyStatesTable, basis, idx1},
    (
      energyStatesTable = BasisTableGenerator[numE];
      basis = Table[
        energyStatesTable[{numE, AllowedJ[numE][[idx1]]}],
        {idx1, 1, Length[AllowedJ[numE]]}];
      basis = Flatten[basis, 1];
      Return[basis]
    )
    ];

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

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

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

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

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

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

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

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

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

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

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

```
      If[StringContainsQ[line,"[DAUGHTER TERM]"],
        daughter=StringSplit[line,"["][[1]];
        CFPTable[[configIndex]]=Append[CFPTable[[configIndex]],{daughter}];
        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];
  )
  ]

  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;
        )
      ];
      normalValues = 1/multiplierValue*col[[2 ;;]];
      Join[{operatorSymbol}, normalValues], {col, tab1[[3 ;;]]}
    ];

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

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

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

```
GenerateThreeBodyTables::usage="This function generates the matrix
    elements for the three body operators using the coefficients of
    fractional parentage, including those beyond f^7.";
Options[GenerateThreeBodyTables] = {"Export" -> False};
GenerateThreeBodyTables[nmax_Integer : 14, OptionsPattern[]] := (
  tiKeys = {"t_{2}", "t_{2}^{'}", "t_{3}", "t_{4}", "t_{6}", "t_{7}",
    "t_{8}", "t_{11}", "t_{11}^{'}", "t_{12}", "t_{14}", "t_{15}",
    "t_{16}", "t_{17}", "t_{18}", "t_{19}"};
  TSymbolsAssoc = AssociationThread[tiKeys -> TSymbols];
  juddOperators = ParseJudd1984[];
  (* op3MatrixElement[SL, SpLp, opSymbol] returns the value for the
    reduced matrix element of the operator opSymbol for the terms {SL,
     SpLp} in the f^3 configuration. *)
  op3MatrixElement[SL_, SpLp_, opSymbol_] := (
    jOP = juddOperators[{3, opSymbol}];
    key = {SL, SpLp};
    val = If[MemberQ[Keys[jOP], key],
      jOP[key],
      0];
    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}
      ]
    ]
    )
  ];

  (* ################# 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.
  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 the a facor proportional to operator z13 is
    subtracted since its effect is seen as 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."
    ;
  ReducedSOOandECSOinf2[SL_, SpLp_] :=
  Module[{a13, z13, z13inf2, matElement, redSOOandECSOinf2},
    a13 = (-33 M0 + 3 M2 + 15/11 M4 -
          6 P0 + 3/2 (35 P2 + 77 P4 + 143 P6));
    z13inf2 = <|
        {"1S","3P"} -> 2,
        {"3P","3P"} -> 1,
        {"3P","1D"} -> -Sqrt[(15/2)],
        {"1D","3F"} -> Sqrt[10],
        {"3F","3F"} -> Sqrt[14],
        {"3F","1G"} -> -Sqrt[11],
        {"1G","3H"} -> Sqrt[10],
        {"3H","3H"} -> Sqrt[55],
        {"3H","1I"} -> -Sqrt[(13/2)]
        |>;
    matElement = Which[
      MemberQ[Keys[z13inf2],{SL,SpLp}],
        z13inf2[{SL,SpLp}],
      MemberQ[Keys[z13inf2],{SpLp,SL}],
        z13inf2[{SpLp,SL}],
      True,
        0
    ];
    redSOOandECSOinf2 = (
        ReducedT11inf2[SL, SpLp] +
        Reducedt11inf2[SL, SpLp] -
        a13 / 6 * matElement
    );
    redSOOandECSOinf2 = SimplifyFun[redSOOandECSOinf2];
    Return[redSOOandECSOinf2];
    ];

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

  GenerateSOOandECSOLSTable::usage="GenerateSOOandECSOLSTable[nmax]
   generates the LS reduced matrix elements of the spin-other-orbit +
    ECSO for the f^n configurations up to n=nmax. The values for n=1
   and n=2 are taken from \"Judd, BR, HM Crosswhite, and Hannah
   Crosswhite. \"Intra-Atomic Magnetic Interactions for f Electrons.\
   " Physical Review 169, no. 1 (1968): 130.\", and the values for n
   >2 are calculated recursively using equation (4) of that same
   paper. The values are then exported to a file \"
   ReducedSOOandECSOLSTable.m\" in the data folder of this module.
   The values are also returned as an association.";
  Options[GenerateSOOandECSOLSTable] = {"Progress" -> True, "Export"
   -> True};
  GenerateSOOandECSOLSTable[nmax_Integer, OptionsPattern[]]:= (
    If[And[OptionValue["Progress"], frontEndAvailable],
      (
        numItersai = Association[Table[numE->Length[AllowedNKSLTerms[
   numE]]^2, {numE, 1, nmax}]];
        counters   = Association[Table[numE->0, {numE, 1, nmax}]];
        totalIters = Total[Values[numItersai[[1;;nmax]]]];
        template1  = StringTemplate["Iteration `numiter` of `totaliter
   `"];
        template2  = StringTemplate["`remtime` min remaining"];
   template3 = StringTemplate["Iteration speed = `speed` ms/it"];
        template4  = StringTemplate["Time elapsed = `runtime` min"];
        progBar = PrintTemporary[
          Dynamic[
            Pane[
              Grid[{
                    {Superscript["f", numE]},
                    {template1[<|"numiter"->numiter, "totaliter"->
   totalIters|>]},
                    {template4[<|"runtime"->Round[QuantityMagnitude[
   UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                    {template2[<|"remtime"->Round[QuantityMagnitude[
   UnitConvert[(Now-startTime)/(numiter)*(totalIters-numiter), "min"
   ]], 0.1]|>]},
                    {template3[<|"speed"->Round[QuantityMagnitude[Now-
   startTime, "ms"]/(numiter), 0.01]|>]}, {ProgressIndicator[Dynamic[
   numiter], {1, totalIters}]}
                  },
                  Frame->All
              ],
              Full,
              Alignment->Center
            ]
          ]
        ];
      )
    ];
    SOOandECSOLSTable = <||>;
    numiter    = 1;
    startTime = Now;
    Do[
      (
        numiter+= 1;
        SOOandECSOLSTable[{numE, SL, SpLp}] = Which[
          numE==1,
          0,
          numE==2,
          SimplifyFun[ReducedSOOandECSOinf2[SL, SpLp]],
          True,
          SimplifyFun[ReducedSOOandECSOinfn[numE,  SL, SpLp]]
        ];
      ),
    {numE, 1, nmax},
    {SL, AllowedNKSLTerms[numE]},
    {SpLp, AllowedNKSLTerms[numE]}
    ];
    If[And[OptionValue["Progress"], frontEndAvailable],
      NotebookDelete[progBar]];
    If[OptionValue["Export"],
      (fname = FileNameJoin[{moduleDir, "data", "
  ReducedSOOandECSOLSTable.m"}];
      Export[fname, SOOandECSOLSTable];
      )
    ];
    Return[SOOandECSOLSTable];
  );

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

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

  ReducedT22inf2::usage="ReducedT22inf2[SL, SpLp] returns the reduced
    matrix element of the scalar component of the double tensor T22
    for the terms SL, SpLp in f^2.
  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,
         0,
         numE==2,
         SimplifyFun[ReducedT22inf2[SL, SpLp]],
         True,
         SimplifyFun[ReducedT22infn[numE,  SL, SpLp]]
       ];
   ),
   {numE, 1, nmax},
   {SL,   AllowedNKSLTerms[numE]},
   {SpLp, AllowedNKSLTerms[numE]}
   ];
   If[And[OptionValue["Progress"],frontEndAvailable],
     NotebookDelete[progBar]
   ];
   If[OptionValue["Export"],
     (
       fname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"
 }];
       Export[fname, T22Table];
     )
   ];
   Return[T22Table];
 );

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

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

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

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

 SOOandECSO::usage="SOOandECSO[n, SL, SpLp, J] returns the matrix
  element <|SL,J|spin-spin|SpLp,J|> for the combined effects of the
  spin-other-orbit interaction and the electrostatically-correlated-
  spin-orbit (which originates from configuration interaction
  effects) within the configuration f^n. This matrix element is
  independent of MJ. This is obtained by querying the relevant
  reduced matrix element by querying the association
  SOOandECSOLSTable and putting in the adequate phase and 6-j symbol
  . The SOOandECSOLSTable puts together the reduced matrix elements
  from three operators.
 This is calculated according to equation (3) in \"Judd, BR, HM
  Crosswhite, and Hannah Crosswhite. \"Intra-Atomic Magnetic
  Interactions for f Electrons.\" Physical Review 169, no. 1 (1968):
   130.\".
 ";
 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, SLJ, SLJp,
  J}] returns the matrix element of the magnetic interaction between
   the terms SLJ and SLJp in the f^n configuration. The interaction
  is given by the sum of the spin-spin interaction and the SOO and
  ECSO interactions. The spin-spin interaction is given by the
  function SpinSpin[{numE, SLJ, SLJp, J}]. The SOO and ECSO
  interactions are given by the function SOOandECSO[{numE, SLJ, SLJp
  , J}]. The function requires chenDeltas to be loaded into the
  session. The option \"ChenDeltas\" can be use to include or
  exclude the Chen deltas from the calculation. The default is to
  exclude them.";
 Options[MagneticInteractions] = {"ChenDeltas" -> False};
 MagneticInteractions[{numE_, SLJ_, SLJp_, J_}, OptionsPattern[]] :=
   (
     key = {numE, SLJ, SLJp, J};
     ss = \[Sigma]SS * SpinSpinTable[key];
     sooandecso = SOOandECSOTable[key];
     total = ss + sooandecso;
     total = SimplifyFun[total];
     If[
       Not[OptionValue["ChenDeltas"]],
       Return[total]
     ];
     (* In the type A errors the wrong values are different *)
     If[MemberQ[Keys[chenDeltas["A"]], {numE, SLJ, SLJp}],
       (
         {S, L} = FindSL[SLJ];
         {Sp, Lp} = FindSL[SLJp];
         phase   = Phaser[Sp + L + J];
         Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
         Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
         {M0v, M2v, M4v, P2v, P4v, P6v} = chenDeltas["A"][{numE, SLJ,
  SLJp}]["wrong"];
         total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
         total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
         total  = total /. Prescaling;
         total  = wChErrA * total + (1 - wChErrA) * (ss + sooandecso)
       )
     ];
     (* In the type B errors the wrong values are zeros all around *)
     If[MemberQ[chenDeltas["B"], {numE, SLJ, SLJp}],
       (
         {S, L} = FindSL[SLJ];
         {Sp, Lp} = FindSL[SLJp];
         phase = Phaser[Sp + L + J];
         Msixjay = SixJay[{Sp, Lp, J},{L, S, 2}];
         Psixjay = SixJay[{Sp, Lp, J},{L, S, 1}];
         {M0v, M2v, M4v, P2v, P4v, P6v} = {0, 0, 0, 0, 0, 0};
         total  = phase * Msixjay(M0v*M0 + M2v*M2 + M4v*M4);
         total += phase * Psixjay(P2v*P2 + P4v*P4 + P6v*P6);
         total  = total /. Prescaling;
         total  = wChErrB * total + (1 - wChErrB) * (ss + sooandecso)
       )
     ];
     Return[total];
   )

```

```
(* ################## 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]
                )
              )
            ]
          )
        ]
      )
    ];
  val
  ]

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[q_, 2] := {Sm22, Sm12, S02,  S12,  S22}[[q + 3]];
Sqk[q_, 4] := {Sm44, Sm34, Sm24, Sm14, S04,  S14,  S24, S34, S44}[[q
   + 5]];
Sqk[q_, 6] := {Sm66, Sm56, Sm46, Sm36, Sm26, Sm16, S06, S16, S26,
  S36, S46, S56, S66}[[q + 7]];

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}
  ]
)

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

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

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

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

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

GSO7W::usage = "GSO7W is an association whose keys are labels for
```

```
    the irreducible representations of group R7 and whose values are
    the eigenvalues of the corresponding Casimir operator.
  Reference: Wybourne, \"Spectroscopic Properties of Rare Earths\",
    table 2-7.";
  GS07W := 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
    }
  ];

  CasimirS07::usage = "CasimirS07[SL, SpLp] returns the LS reduced
    matrix element of the configuration interaction term corresponding
     to the Casimir operator of R7.";
  CasimirS07[{SL_, SpLp_}] := (
    Wlabel = FindNKLSTerm[SL][[1]][[3]];
    If[SL==SpLp,
      γ * GS07W[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,
        CasimirS03[{SL, SL}] +
        CasimirS07[{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};
  TabulateJJBlockMatrixTable[numE_, CFTable_, OptionsPattern[]]:= (
    JJBlockMatrixTable = <||>;
    totalIterations = Length[AllowedJ[numE]]^2;
    template1 = StringTemplate["Iteration `numiter` of `totaliter`"];
    template2 = StringTemplate["`remtime` min remaining"];
    template4 = StringTemplate["Time elapsed = `runtime` min"];
    numiter   = 0;
    startTime = Now;
    If[$FrontEnd =!= Null,
      (
        temp = PrintTemporary[
          Dynamic[
            Grid[
              {
                {template1[<|"numiter"->numiter, "totaliter"->totalIterations|>]},
                {template2[<|"remtime"->Round[QuantityMagnitude[UnitConvert[(Now-startTime)/(Max[1,numiter])*(totalIterations-numiter), "min"]], 0.1]|>]},
                {template4[<|"runtime"->Round[QuantityMagnitude[UnitConvert[(Now-startTime), "min"]], 0.1]|>]},
                {ProgressIndicator[numiter, {1, totalIterations}]}
              }
            ]
        ]
        ];
      )
    ];
    Do[
      (
        JJBlockMatrixTable[{numE, J, Jp}] = JJBlockMatrix[numE, J, Jp, CFTable, "Sparse"->OptionValue["Sparse"], "ChenDeltas" -> OptionValue["ChenDeltas"]];
        numiter += 1;
      ),
    {Jp, AllowedJ[numE]},
    {J, AllowedJ[numE]}
    ];
    If[$FrontEnd =!= Null,
      NotebookDelete[temp]
    ];
    Return[JJBlockMatrixTable];
  )

  TabulateManyJJBlockMatrixTables::usage = "TabulateManyJJBlockMatrixTables[{n1, n2, ...}] calculates the tables of matrix elements for the requested f^n_i configurations. The function does not return the matrices themselves. It instead returns an association whose keys are numE and whose values are the filenames where the output of TabulateJJBlockMatrixTables was saved to.The output consists of an association whose keys are of the form {n, J, Jp} and whose values are rectangular arrays given the values of <|LSJMJa|H|L'S'J'MJ'a'|>.";
  Options[TabulateManyJJBlockMatrixTables] = {"Overwrite"->False, "Sparse"->True, "ChenDeltas"->False, "FilenameAppendix"-> "", "Compressed" -> False};
  TabulateManyJJBlockMatrixTables[ns_, OptionsPattern[]]:= (
    overwrite = OptionValue["Overwrite"];
    fNames = <||>;
    fileApp = OptionValue["FilenameAppendix"];
    ExportFun = If[OptionValue["Compressed"], ExportMZip, Export];
    Do[
      (
        CFdataFilename = FileNameJoin[{moduleDir, "data", "CrystalFieldTable_f"<>ToString[numE]<>".zip"}];
        PrintTemporary["Importing CrystalFieldTable from ", CFdataFilename, " ..."];
  CrystalFieldTable = ImportMZip[CFdataFilename];

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

  HamMatrixAssembly::usage="HamMatrixAssembly[numE] returns the
    Hamiltonian matrix for the f^n_i configuration. The matrix is
    returned as a SparseArray. 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;";
  Options[HamMatrixAssembly] = {"FilenameAppendix"->"","IncludeZeeman"
    ->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];
     ),
     params = params0;
   ];
   (* 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 *)
   blockHam = ArrayFlatten[blockHam];
   blockHam = ReplaceInSparseArray[blockHam, params];
   If[OptionValue["IncludeZeeman"],
     (
      PrintTemporary["Including Zeeman terms ..."];
      {magx, magy, magz} = MagDipoleMatrixAssembly[numE];
      blockHam += - TeslaToKayser * (Bx * magx + By * magy + Bz *
    magz);
     )
   ];
   blockHam = MapToSparseArray[blockHam, Expand];
   Return[blockHam];
   ]

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 exports
    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.";
Options[SimplerSymbolicHamMatrix]={
  "Export"->True,
  "PrependToFilename"->"",
  "EorF"->"F",
  "Overwrite" -> False,
  "Return" -> True,
  "IncludeZeeman" -> False};
SimplerSymbolicHamMatrix[numE_Integer, simplifier_List, OptionsPattern
    []]:=Module[
  {thisHam,eTofs,fname},
  (
    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"]<>"SymbolicMatrix-f"<>ToString[numE]<>".m"}];
    If[FileExistsQ[fname] && Not[OptionValue["Overwrite"]],
      (
        If[OptionValue["Return"],
          (
            Print["File ",fname," already exists, and option \"
    Overwrite\" is set to False, loading file ..."];
            thisHam = Import[fname];
            Return[thisHam];
          ),
          (
            Print["File ",fname," already exists, skipping ..."];
          Return[Null];
          )
        ]
      )
    ];

    thisHam=HamMatrixAssembly[numE, "IncludeZeeman"->OptionValue["
    IncludeZeeman"]];
    thisHam=ReplaceInSparseArray[thisHam, simplifier];
    If[OptionValue["Export"],
    (
      Print["Exporting to file ",fname];
      Export[fname,thisHam]
    )
    ];
    If[OptionValue["Return"],
      Return[thisHam],
      Return[Null]
    ];
  )
]

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

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

  magOp = <||>;

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

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

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

  MagDipoleMatrixAssembly::usage="MagDipoleMatrixAssembly[numE]
   returns the matrix representation of the operator - 1/2 (L + gs S)
    in the f^numE configuration. The function returns a list with
   three elements corresponding to the x,y,z components of this
   operator. The option \"FilenameAppendix\" can be used to append a
   string to the filename from which the function imports from in
   order to patch together the array. For numE beyond 7 the function
   returns the same as for the complementary configuration.";
  Options[MagDipoleMatrixAssembly]={"FilenameAppendix"->""};
  MagDipoleMatrixAssembly[nf_,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}
    ];
    blockOp = ArrayFlatten[blockOp];
    opMinus = blockOp[[;; , ;; , 1]];
    opZero =  blockOp[[;; , ;; , 2]];
    opPlus =  blockOp[[;; , ;; , 3]];
    opX =   (opMinus - opPlus)/Sqrt[2];
    opY = I (opPlus + opMinus)/Sqrt[2];
    opZ = opZero;
    Return[{opX, opY, opZ}];
  )
  ];

  MagDipLineStrength::usage="MagDipLineStrength[theEigensys, numE]
   takes the eigensystem of an ion and the number numE of f-electrons
    that correspond to it and it calculates the line strength array
   Stot.
  The option \"Units\" can be set to either \"SI\" (so that the units
   of the returned array are A/m^2) or to \"Hartree\".
  The option \"States\" can be used to limit the states for which the
   line strength is calculated. The default, All, calculates the line
    strength for all states. A second option for this is to provide
   an index labelling a specific state, in which case only the line
   strengths between that state and all the others are computed.
  The returned array should be interpreted in the eigenbasis of the
   Hamiltonian. As such the element Stot[[i,i]] corresponds to the
   line strength states |i> and |j>.";
  Options[MagDipLineStrength]={"Reload MagOp" -> False, "Units"->"SI",
    "States" -> All};
  MagDipLineStrength[theEigensys_List, numE0_Integer, OptionsPattern
    []]:=Module[
    {allEigenvecs, Sx, Sy, Sz, Stot ,factor},
  (
    numE = Min[14-numE0, numE0];
    (*If not loaded then load it, *)
    If[Or[
        Not[MemberQ[Keys[magOp], numE]],
        OptionValue["Reload MagOp"]],
    (
      magOp[numE] = ReplaceInSparseArray[#,{gs->2}]& /@
   MagDipoleMatrixAssembly[numE];
    )
    ];
    allEigenvecs = Transpose[Last/@theEigensys];
    Which[OptionValue["States"] === All,
      (
        {Sx,Sy,Sz}   = (ConjugateTranspose[allEigenvecs].#.
   allEigenvecs) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      ),
      IntegerQ[OptionValue["States"]],
      (
        singleState  = theEigensys[[OptionValue["States"],2]];
        {Sx,Sy,Sz}   = (ConjugateTranspose[allEigenvecs].#.singleState
   ) & /@ magOp[numE];
        Stot         = Abs[Sx]^2+Abs[Sy]^2+Abs[Sz]^2;
      )
    ];
    Which[
      OptionValue["Units"] == "SI",
        Return[4 \[Mu]B^2 * Stot],
      OptionValue["Units"] == "Hartree",
        Return[Stot],
      True,
      (
        Print["Invalid option for \"Units\". Options are \"SI\" and \"
   Hartree\"."];
        Abort[];
      )
    ];
  )
  ]

  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. 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>.";
  Options[MagDipoleRates]={"Units"->"SI", "Lifetime"->False};
  MagDipoleRates[eigenSys_List, numE0_Integer,OptionsPattern[]]:=
   Module[
    {AMD,gKramers,Stot,eigenEnergies,transitionWaveLengthsInMeters},(
    numE          = Min[14-numE0, numE0];
    gKramers      = If[OddQ[numE],2,1];
    Stot          = MagDipLineStrength[eigenSys, numE, "Units"->
   OptionValue["Units"]];
    eigenEnergies = Chop[First/@eigenSys];
    energyDiffs   = Outer[Subtract,eigenEnergies,eigenEnergies];
    energyDiffs   = ReplaceDiagonal[energyDiffs, Indeterminate];
    (* Energies assumed in pseudo-energy unit kayser.*)
    transitionWaveLengthsInMeters = 0.01/energyDiffs;

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

  GroundStateOscillatorStrength::usage="GroundStateOscillatorStrength[
   eigenSys, numE] calculates the oscillator strength between the
   ground state and the excited states as given by eigenSys. The
   energy unit assumed in eigenSys is kayser. The returned array
   should be interpreted in the eigenbasis of the Hamiltonian. As
   such the element fMDGS[[i]] corresponds to the oscillator strength
    between ground state and eigenstate |i>.";
  GroundStateOscillatorStrength[eigenSys_, numE_]:=Module[
  {eigenEnergies, SMDGS, GSEnergy, gKramers, energyDiffs,
   transitionWaveLengthsInMeters, factor},
  (
    eigenEnergies     = First/@eigenSys;
    SMDGS             = MagDipLineStrength[eigenSys,numE, "Units"->"SI"
   , "States"->1];
    GSEnergy          = eigenSys[[1,1]];
    gKramers          = If[OddQ[numE],2,1];
    energyDiffs       = eigenEnergies-GSEnergy;
    energyDiffs[[1]] = Indeterminate;
    transitionWaveLengthsInMeters = 0.01/energyDiffs;
    factor = (8\[Pi]^2 me)       /(3 hPlanck eCharge^2 cLight);
    fMDGS=unitFactor/gKramers/transitionWaveLengthsInMeters*SMDGS;
    Return[fMDGS];
  )
  ]
```

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

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

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

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

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

PrintSLJM::usage = "Given a list with four elements {S, L, J, MJ}
  this function returns a symbol where the spin multiplicity is
  presented as a superscript, the orbital angular momentum as its
  corresponding spectroscopic letter, and {J, MJ} as a subscript. No
   attempt is made to guarantee that the given input is consistent."
  ;
PrintSLJM[SLJM_] :=
  RowBox[{SuperscriptBox[" ", 2 SLJM[[1]] + 1],
    SubscriptBox[PrintL[SLJM[[2]]], {SLJM[[3]], SLJM[[4]]}]}] //
  DisplayForm;

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

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

AllowedSLTerms::usage = "AllowedSLTerms[numE] returns a list with
  the allowed terms in the f^numE configuration, the terms are given
   as lists in the format {S, L}. This list may have redundancies
  which are compatible with the degeneracies that might correspond
  to the given case.";
AllowedSLTerms[numE_] := Map[FindSL[First[#]] &, CFPTerms[Min[numE,
  14-numE]]]

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

MaxJ::usage = "MaxJ[numE] gives the maximum J = S+L that corresponds
   to the configuration f^numE.";
MaxJ[numE_] := Max[Map[Total, AllowedSLTerms[Min[numE, 14-numE]]]]

MinJ::usage = "MinJ[numE] gives the minimum J = S+L that corresponds
   to the configuration f^numE.";
MinJ[numE_] := Min[Map[Abs[Part[#, 1] - Part[#, 2]] &,
  AllowedSLTerms[Min[numE, 14-numE]]]]

AllowedSLJTerms::usage = "AllowedSLJTerms[numE] returns a list with
  the allowed {S, L, J} terms in the f^n configuration, the terms
  are given as lists in the format {S, L, J}. This list may have
  repeated elements which account for possible degeneracies of the
  related term.";
AllowedSLJTerms[numE_] :=
  Module[{idx1, allowedSL, allowedSLJ},
    allowedSL = AllowedSLTerms[numE];
    allowedSLJ = {};
    For[
      idx1 = 1,
      idx1 <= Length[allowedSL],
      termSL = allowedSL[[idx1]];
      termsSLJ =
        Table[
          {termSL[[1]], termSL[[2]], J},
        {J,Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
        ];
      allowedSLJ = Join[allowedSLJ, termsSLJ];
      idx1++
    ];
    SortBy[allowedSLJ, Last]
  ]

AllowedNKSLJTerms::usage = "AllowedNKSLJTerms[numE] returns a list
  with the allowed {SL, J} terms in the f^n configuration, the terms
   are given as lists in the format {SL, J} where SL is a string in
  spectroscopic notation.";
AllowedNKSLJTerms[numE_] :=
  Module[{allowedSL, allowedNKSL, allowedSLJ, nn},
    allowedNKSL = AllowedNKSLTerms[numE];
    allowedSL = AllowedSLTerms[numE];
    allowedSLJ = {};
    For[
      nn = 1,
      nn <= Length[allowedSL],
      (
        termSL = allowedSL[[nn]];
        termNKSL = allowedNKSL[[nn]];
        termsSLJ =
          Table[{termNKSL, J},
          {J, Abs[termSL[[1]] - termSL[[2]]], Total[termSL]}
          ];
        allowedSLJ = Join[allowedSLJ, termsSLJ];
        nn++
      )
    ];
    SortBy[allowedSLJ, Last]
  ]

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.";
AllowedNKSLforJTerms[numE_, J_] := Module[
    {allowedSL, allowedNKSL, allowedSLJ, nn, termSL, termNKSL,
  termsSLJ},
    allowedNKSL = AllowedNKSLTerms[numE];
    allowedSL = AllowedSLTerms[numE];
    allowedSLJ = {};
    For[
      nn = 1,
      nn <= Length[allowedSL],
      (
        termSL = allowedSL[[nn]];
        termNKSL = allowedNKSL[[nn]];
        termsSLJ = If[Abs[termSL[[1]] - termSL[[2]]] <= J <= Total[
  termSL],
          {{termNKSL, J}},
          {}
          ];
        allowedSLJ = Join[allowedSLJ, termsSLJ];
        nn++
      )
    ];
    Return[allowedSLJ]
  ];

AllowedSLJMTerms::usage = "AllowedSLJMTerms[numE] returns a list
  with all the states that correspond to the configuration f^n. A
  list is returned whose elements are lists of the form {S, L, J, MJ
  }.";
AllowedSLJMTerms[numE_] := Module[
    {allowedSLJ, allowedSLJM, termSLJ, termsSLJM, nn},
    allowedSLJ = AllowedSLJTerms[numE];
    allowedSLJM = {};
    For[
      nn = 1,
      nn <= Length[allowedSLJ],
      nn++,
      (
        termSLJ = allowedSLJ[[nn]];
        termsSLJM =
          Table[{termSLJ[[1]], termSLJ[[2]], termSLJ[[3]], M},
          {M, - termSLJ[[3]], termSLJ[[3]]}
          ];
        allowedSLJM = Join[allowedSLJM, termsSLJM];
      )
    ];
    Return[SortBy[allowedSLJM, Last]];
    ]

AllowedNKSLJMforJMTerms::usage = "AllowedNKSLJMforJMTerms[numE, J,
  MJ] returns a list with all the terms that contain states of the f
  ^n configuration that have a total angular momentum J, and a
  projection along the z-axis MJ. The returned list has elements of
  the form {SL (string in spectroscopic notation), J, MJ}.";
AllowedNKSLJMforJMTerms[numE_, J_, MJ_] :=
  Module[{allowedSL, allowedNKSL, allowedSLJM, nn},
    allowedNKSL = AllowedNKSLTerms[numE];
    allowedSL   = AllowedSLTerms[numE];
    allowedSLJM = {};
    For[
      nn = 1,
      nn <= Length[allowedSL],
      termSL = allowedSL[[nn]];
      termNKSL = allowedNKSL[[nn]];
      termsSLJ = If[(Abs[termSL[[1]] - termSL[[2]]]
                    <= J
                    <= Total[termSL]
                    && (Abs[MJ] <= J)
                    ),
                    {{termNKSL, J, MJ}},
                    {}];
      allowedSLJM = Join[allowedSLJM, termsSLJ];
      nn++
    ];
    Return[allowedSLJM];
  ]

AllowedNKSLJMforJTerms::usage = "AllowedNKSLJMforJTerms[numE, J]
  returns a list with all the states that have a total angular
  momentum J. The returned list has elements of the form {{SL (
  string in spectroscopic notation), J}, MJ}, 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
      };
  LoadParameters[Ln_String, OptionsPattern[]]:=
    Module[{source, params},
    (
      source = OptionValue["Source"];
      params = Which[source=="Carnall",
              (Association[Carnall["data"][Ln]])
              ];
      (*If a free ion then all the parameters from the crystal field
    are set to zero*)
      If[OptionValue["Free Ion"],
        Do[params[cfSymbol] = 0,
        {cfSymbol, cfSymbols}
        ]
      ];
      params[F0] = 0;
      params[M2] = 0.56 * params[M0]; (* See Carnall 1989, Table I,
    caption, probably fixed based on HF values*)
      params[M4] = 0.31 * params[M0]; (* See Carnall 1989, Table I,
    caption, probably fixed based on HF values*)
      params[P0] = 0;
      params[P4] = 0.5 * params[P2];  (* See Carnall 1989, Table I,
    caption, probably fixed based on HF values*)
      params[P6] = 0.1 * params[P2];  (* See Carnall 1989, Table I,
    caption, probably fixed based on HF values*)
      params[gs] = OptionValue["gs"];
      {params[E0], params[E1], params[E2], params[E3]} = FtoE[params[
    F0], params[F2], params[F4], params[F6]];
      params[E0] = 0;
      Return[params];
    )
  ];

  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];
      )
    ]

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.
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.
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\".
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.
Tha ordered basis in which this eigenvector is to be interpreted is
  the one corresponding to BasisLSJMJ[numE].
The function admits the following options:
\"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.
\"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.
\"Zeroes\" (list) : A list with symbols assumed to be zero.
";
Options[IonSolver] = {"Include Spin-Spin" -> True,
   "Overwrite Hamiltonian" -> False,
   "Zeroes" -> {}};
IonSolver[numE_Integer, params0_Association, host_String:"Ln",
  OptionsPattern[]] := Module[
   {ln, simplifier, simpleHam, numHam, eigensys,
   startTime, endTime, diagonalTime, params=params0, zeroSymbols},
(
   ln = StringSplit["Ce Pr Nd Pm Sm Eu Gd Tb Dy Ho Er Tm Yb"][[numE
  ]];

   (* This could be done when replacing values, but this produces
  smaller saved arrays. *)
   simplifier = (#-> 0) & /@ OptionValue["Zeroes"];
   simpleHam = SimplerSymbolicHamMatrix[numE,
     simplifier,
     "PrependToFilename" -> host,
     "Overwrite" -> OptionValue["Overwrite Hamiltonian"]
   ];

   (* Note that we don't have to flip signs of parameters for fn
  beyond f7 since the matrix produced
   by SimplerSymbolicHamMatrix has already accounted for this. *)

   (* Everything that is not given is set to zero *)
   params = ParamPad[params, "Print" -> True];
   PrintFun[params];

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

   (* Create the numeric hamiltonian *)
   numHam = ReplaceInSparseArray[simpleHam, params];
   Clear[simpleHam];
```

```

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

    (* Shift the baseline energy *)
    eigensys = ShiftedLevels[eigensys];
    (* Sort according to energy *)
    eigensys = SortBy[eigensys, First];
    Return[eigensys];
    )
  ]


  ShiftedLevels::usage = "
  ShiftedLevels[originalLevels] takes a list of levels of the form
  {{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.";
  ShiftedLevels[originalLevels_] :=
    Module[{groundEnergy, shifted},
      groundEnergy = Sort[originalLevels][[1,1]];
      shifted      = Map[{#[[1]] - groundEnergy, #[[2]]} &,
    originalLevels];
      Return[shifted];
    ]

  (* ########################################################### *)
  (* ################## 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]
    )

  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[states, 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. 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[states
    , basis, numBasisVecs, roundTo] 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
    at most numBasisVecs basis vectors. By default roundTo is 0.01 and
     this is the value used to round the amplitude coefficients.
  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 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+1]]
    ]

  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*)
    orderedProbs       = probs[[ordering]];
    (*To speed up Accumulate, assume that only as much as maxParts
    will be needed*)
    truncationIndex    = Min[maxParts, Length[orderedProbs]];
    orderedProbs       = orderedProbs[[;;truncationIndex]];
    (*Accumulate the probabilities*)
    accProb            = Accumulate[orderedProbs];
    (*Find the index of the first element in accProb that is greater
    than probSum*)
    thresholdIndex     = Min[Length[accProb], FindThresholdPosition[
    accProb, probSum]];
    (*Grab all the indicees up till that one*)
    chosenIndices      = ordering[[;; thresholdIndex]];
    (*Select the corresponding elements from the basis*)
    majorComponents    = basis[[chosenIndices]];
    (*Select the corresponding amplitudes*)
    majorAmplitudes    = amplitudes[[chosenIndices]];
    (*Take their absolute value*)
    absMajorAmplitudes = Abs[majorAmplitudes];
    (*Make sure that there are no effectively zero contributions*)
    notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_ /; x
    != 0]];
    (* majorComponents    = PrettySaundersSLJmJ
    [{#[[1]],#[[2]],#[[3]]}] & /@ majorComponents; *)
    majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
    majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
    (*Make them into Kets*)
    majorComponents    = Ket /@ majorComponents[[notnullAmplitudes]];
    (*Multiply and add to build the final Ket*)
    majorRep           = majorAmplitudes . majorComponents;
      );
  Return[{energy, majorRep}]
  ],
    CompilationTarget -> "C",
    RuntimeAttributes -> {Listable},
    Parallelization -> True,
    RuntimeOptions -> "Speed"
  ];

  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.";
  ParseStatesByProbabilitySum[eigensys_, basis_, probSum_, roundTo_ : 0.01, maxParts_: 20] := Module[
    {parsedByProb, numStates, state, energy, eigenVec, amplitudes, probs, ordering,
    orderedProbs, truncationIndex, accProb, thresholdIndex, chosenIndices, majorComponents,
    majorAmplitudes, absMajorAmplitudes, notnullAmplitudes, majorRep},
  (
    numStates    = Length[eigensys];
    parsedByProb = Table[(
      state              = eigensys[[idx]];
      {energy, eigenVec} = state;
      (*Round them up*)
      amplitudes         = Round[eigenVec, roundTo];
      probs              = Round[Abs[eigenVec^2], roundTo];
      ordering           = Reverse[Ordering[probs]];
      (*Order the probabilities from high to low*)
      orderedProbs       = probs[[ordering]];
      (*To speed up Accumulate, assume that only as much as maxParts will be needed*)
      truncationIndex    = Min[maxParts, Length[orderedProbs]];
      orderedProbs       = orderedProbs[[;;truncationIndex]];
      (*Accumulate the probabilities*)
      accProb            = Accumulate[orderedProbs];
      (*Find the index of the first element in accProb that is greater than probSum*)
      thresholdIndex     = Min[Length[accProb], FindThresholdPosition[accProb, probSum]];
      (*Grab all the indicees up till that one*)
      chosenIndices      = ordering[[;; thresholdIndex]];
      (*Select the corresponding elements from the basis*)
      majorComponents    = basis[[chosenIndices]];
      (*Select the corresponding amplitudes*)
      majorAmplitudes    = amplitudes[[chosenIndices]];
      (*Take their absolute value*)
      absMajorAmplitudes = Abs[majorAmplitudes];
      (*Make sure that there are no effectively zero contributions*)
      notnullAmplitudes  = Flatten[Position[absMajorAmplitudes, x_ /; x != 0]];
      (* majorComponents    = PrettySaundersSLJmJ[{#[[1]],#[[2]],#[[3]]}} & /@ majorComponents; *)
      majorComponents    = PrettySaundersSLJmJ /@ majorComponents;
      majorAmplitudes    = majorAmplitudes[[notnullAmplitudes]];
      majorComponents    = majorComponents[[notnullAmplitudes]];
      (*Multiply and add to build the final Ket*)
      majorRep           = majorAmplitudes . majorComponents;
      {energy, majorRep}
      ), {idx, numStates}];
  Return[parsedByProb]
  )
  ];


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

  (* ############################################################## *)
  (* ########################## Misc ########################## *)

  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.";
  SymbToNum[expr_, replacementAssociation_]:= (
    includedKeys = Keys[replacementAssociation];
    (*If a key is not defined, make its value zero.*)
    fullAssociation = Table[(
      If[MemberQ[includedKeys, key],
        ToExpression[key]->replacementAssociation[key],
        ToExpression[key]->0
      ]
    ),
    {key, paramSymbols}];
    Return[expr/.fullAssociation];
  )

  SimpleConjugate::usage = "SimpleConjugate[expr] takes an expression
    and applies a simplified version of the conjugate in that all it
    does is that it replaces the imaginary unit I with -I. It assumes
    that every other symbol is real so that it remains the same under
    complex conjugation. Among other expressions it is valid for any
    rational or polynomial expression with complex coefficients and
    real variables.";
  SimpleConjugate[expr_] := expr /. Complex[a_, b_] :> a - I b;

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

  ImportMZip::usage="ImportMZip[filename] imports a .m file inside a .
    zip file with corresponding filename. If the Option \"Leave
    Uncompressed\" is set to True (the default) then this function
    also leaves an umcompressed version of the object in the same
    folder of filename";
  Options[ImportMZip]={"Leave Uncompressed" -> True};
  ImportMZip[filename_String, OptionsPattern[]] := Module[
    {baseName, importKey, zipImportName, mImportName, imported},
  (
    baseName      = FileBaseName[filename];
    (*Function allows for the filename to be .m or .zip*)
    importKey     = baseName <> ".m";
    zipImportName = StringReplace[filename, ".m"->".zip"];
    mImportName   = StringReplace[zipImportName, ".zip"->".m"];
    If[FileExistsQ[mImportName],
    (
      PrintTemporary[".m version exists already, importing that
    instead ..."];
      Return[Import[mImportName]];
    )
    ];
    imported = Import[zipImportName, importKey];
    If[OptionValue["Leave Uncompressed"],
      Export[mImportName, imported]
    ];
    Return[imported]
  )
  ];

  ReplaceInSparseArray::usage = "ReplaceInSparseArray[sparseArray,
   rules] takes a sparse array that may contain symbolic quantities
   and returns a sparse array in which the given replacement rules
   have been used.";
  ReplaceInSparseArray[s_SparseArray, rule_] := (With[{
      elem = s["NonzeroValues"]/.rule,
      def  = s["Background"]/.rule
      },
      srep = SparseArray[Automatic,
        s["Dimensions"],
        def,
        {1, {s["RowPointers"], s["ColumnIndices"]}, elem}
        ];
    ];
    Return[srep];
    );

  MapToSparseArray::usage = "MapToSparseArray[sparseArray, function]
   takes a sparse array and returns a sparse array after the function
    has been applied to it.";
  MapToSparseArray[sparsey_SparseArray, func_] := Module[{
      nonZ, backg, mapped
      },
     (
      nonZ   = func/@ sparsey["NonzeroValues"];
      backg  = func[sparsey["Background"]];
      mapped = SparseArray[Automatic,
        sparsey["Dimensions"],
        backg,
        {1, {sparsey["RowPointers"], sparsey["ColumnIndices"]}, nonZ}
        ];
      Return[mapped];
     )
    ];

  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.";
  Options[ParseTeXLikeSymbol] = {"Form" -> "List"};
  ParseTeXLikeSymbol[bigString_, OptionsPattern[]] := (
    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]
      ]
    );

  (* ########################## Misc ########################## *)
  (* ########################################################### *)

  (* ########################################################### *)
  (* ################# Some Plotting Routines ################# *)

  EnergyLevelDiagram::usage = "EnergyLevelDiagram[states] takes states
     and produces a visualization of its energy spectrum.
  The resultant visualization can be navigated by clicking and
    dragging to zoom in on a region, or by clicking and dragging
    horizontally while pressing Ctrl. Double-click to reset the view."
    ;
  Options[EnergyLevelDiagram] = {
    "Title"->"",
    "ImageSize"->1000,
    "AspectRatio" -> 1/8,
    "Background"->"Automatic",
    "Epilog"->{},
    "Explorer"->True
    };
  EnergyLevelDiagram[states_, OptionsPattern[]]:= (
    energies = First/@states;
    epi = OptionValue["Epilog"];
    explora = If[OptionValue["Explorer"],
      ExploreGraphics,
      Identity
    ];
    explora@ListPlot[Tooltip[{{#, 0}, {#, 1}}, {Quantity[#/8065.54429,
     "eV"], Quantity[#, 1/"Centimeters"]}] &/@ energies,
      Joined        -> True,
      PlotStyle     -> Black,
      AspectRatio   -> OptionValue["AspectRatio"],
      ImageSize     -> OptionValue["ImageSize"],
      Frame         -> True,
      PlotRange     -> {All, {0, 1}},
      FrameTicks    -> {{None, None}, {Automatic, Automatic}},
      FrameStyle    -> Directive[15, Dashed, Thin],
      PlotLabel     -> Style[OptionValue["Title"], 15, Bold],
      Background    -> OptionValue["Background"],
      FrameLabel    -> {"\!\(\*FractionBox[\(E\), SuperscriptBox[\(cm\)
    , \(-1\)]]\)"},
      Epilog        -> epi]
  )

  ExploreGraphics::usage =
    "Pass a Graphics object to explore it. Zoom by clicking and
    dragging a rectangle. Pan by clicking and dragging while pressing
    Ctrl. Click twice to reset view.
    Based on ZeitPolizei @ https://mathematica.stackexchange.com/
    questions/7142/how-to-manipulate-2d-plots";

  OptAxesRedraw::usage =
    "Option for ExploreGraphics to specify redrawing of axes. Default
    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};
          )}]]];

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

```
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.";
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"]]
    )

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},
             {j - 1, numRows - i} + {1, 1}
             ]
           },
         Row[{braLabels[[i]],ketLabelsUpright[[j]],"=",hamMatrix[[i,
  j]]}]
         ],
       {i, 1, numRows},
       {j, 1, numCols}
       ]
     ],
     And[OptionValue["Hover"], OptionValue["Overlay Values"]],
     Flatten[
       Table[
         Tooltip[
           {
             Transparent,
             Rectangle[
             {j - 1, numRows - i},
             {j - 1, numRows - i} + {1, 1}
             ]
           },
         DisplayForm[RowBox[{"\[LeftAngleBracket]", basisLabels[[i]],
   "\[LeftBracketingBar]", basisLabels[[j]], "\[RightAngleBracket]"
  }]]
         ],
       {i, numRows},
       {j, numCols}
       ]
     ],
     True,
     {}
     ];
   textOverlay = If[OptionValue["Overlay Values"],
     (
       Flatten[
       Table[
         Text[hamMatrix[[i, j]],
           {j - 1/2, numRows - i + 1/2}
         ],
       {i, 1, numRows},
       {j, 1, numCols}
       ]
       ]
     ),
     {}
     ];
   epiThings = Join[epiThings, textOverlay];
   MatrixPlot[hamMatrix,
     FrameTicks -> {
       {Transpose[{Range[Length[braLabels]], braLabels}], None},
       {None, Transpose[{Range[Length[ketLabels]], ketLabels}]}
       },
     Evaluate[FilterRules[{opts}, Options[MatrixPlot]]],
     Epilog -> epiThings
   ]
   );

(* ################# Some Plotting Routines ################# *)
(* ########################################################### *)

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

LoadAll::usage="LoadAll[] executes all Load* functions.";
LoadAll[]:=(
     LoadTermLabels[];
     LoadCFP[];
     LoadUk[];
     LoadV1k[];
     LoadT22[];
     LoadSOOandECSOLS[];

     LoadElectrostatic[];
     LoadSpinOrbit[];
     LoadSOOandECSO[];
     LoadSpinSpin[];
     LoadThreeBody[];
     LoadChenDeltas[];
     LoadCarnall[];
   )

fnTermLabels::usage = "This list contains the labels of f^n configurations. Each element of the list has four elements {LS, seniority, W, U}. At first sight this seems to only include the labels for the f^6 and f^7 configuration, however, all is included in these two.";

LoadTermLabels::usage="LoadTermLabels[] loads into the session the labels for the terms in the f^n configurations.";
LoadTermLabels[]:= (
   If[ValueQ[fnTermLabels], Return[]];
   PrintTemporary["Loading data for state labels in the f^n configurations..."];
   fnTermsFname = FileNameJoin[{moduleDir, "data", "fnTerms.m"}];

   If[!FileExistsQ[fnTermsFname],
     (PrintTemporary[">> fnTerms.m not found, generating ..."];
       fnTermLabels = ParseTermLabels["Export"->True];
     ),
     fnTermLabels = Import[fnTermsFname];
   ];
)


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];
   ];
)

LoadChenDeltas::usage="LoadChenDeltas[] loads the differences noted by Chen.";
LoadChenDeltas[]:=(
   If[ValueQ[chenDeltas], Return[]];
   PrintTemporary["Loading the association of discrepancies found by Chen ..."];
   chenDeltasFname = FileNameJoin[{moduleDir, "data", "chenDeltas.m"}];
   If[!FileExistsQ[chenDeltasFname],
     (PrintTemporary[">> chenDeltas.m not found, generating ..."];
       chenDeltas = ParseChenDeltas[];
     ),
     chenDeltas = Import[chenDeltasFname];
   ];
);

ParseChenDeltas::usage="ParseChenDeltas[] parses the data found in ./data/the-chen-deltas-A.csv and ./data/the-chen-deltas-B.csv. If the option \"Export\" is set to True (True is the default), then the parsed data is saved to ./data/chenDeltas.m";
Options[ParseChenDeltas] = {"Export" -> True};
ParseChenDeltas[OptionsPattern[]]:=(
   chenDeltasRaw = Import[FileNameJoin[{moduleDir, "data", "the-chen-deltas-A.csv"}]];
   chenDeltasRaw = chenDeltasRaw[[2 ;;]];
```

```
    chenDeltas = <||>;
    chenDeltasA = <||>;
    Off[Power::infy];
    Do[
      ({right, wrong} = {chenDeltasRaw[[row]][[4 ;;]],
        chenDeltasRaw[[row + 1]][[4 ;;]]};
      key = chenDeltasRaw[[row]][[1 ;; 3]];
      repRule = (#[[1]] -> #[[2]]*#[[1]]) & /@
        Transpose[{{M0, M2, M4, P2, P4, P6}, right/wrong}];
      chenDeltasA[key] = <|"right" -> right, "wrong" -> wrong,
        "repRule" -> repRule|>;
      chenDeltasA[{key[[1]], key[[3]], key[[2]]}] = <|"right" -> right
  ,
        "wrong" -> wrong, "repRule" -> repRule|>;
      ),
      {row, 1, Length[chenDeltasRaw], 2}];
    chenDeltas["A"] = chenDeltasA;

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

  ParseCarnall::usage="ParseCarnall[] parses the data found in ./data/
    Carnall.xls. If the option \"Export\" is set to True (True is the
    default), then the parsed data is saved to ./data/Carnall. This
    data is from the tables and appendices of Carnall et al (1989).";
  Options[ParseCarnall] = {"Export" -> True};
  ParseCarnall[] := (
    ions          = {"Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho","Er",
  "Tm"};
    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, {91, 182, 1001, 1001, 3003,
     1716, 3003, 1001, 1001, 182, 91}];
    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"}]][[i]];
    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[[i-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;
```

```
      ),
  {i,4,14}
  ];

  goodions = Select[ions,#!="Pm"&];
  expData  = Select[Transpose[Carnall["appendix:"<>#<>":RawTable"
 ]][[1+Position[Carnall["appendix:"<>#<>":Headings"],"Exp (1/cm)"
 ][[1,1]]]],NumberQ]&/@goodions;
  Carnall["All Experimental Data"]=AssociationThread[goodions,
 expData];
  If[OptionValue["Export"],
    (
      carnallFname = FileNameJoin[{moduleDir, "data", "Carnall.m"}];
      Print["Exporting to "<>exportFname];
      Export[carnallFname, Carnall];
    )
    ];
  Return[Carnall];
)

CFP::usage = "CFP[{n, NKSL}] provides a list whose first element
 echoes NKSL and whose other elements are lists with two elements
 the first one being the symbol of a parent term and the second
 being the corresponding coefficient of fractional parentage. n
 must satisfy 1 <= n <= 7";

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[].";

LoadCFP::usage="LoadCFP[] loads CFP, CFPAssoc, and CFPTable into the
  session.";
LoadCFP[]:=(
  If[And[ValueQ[CFP], ValueQ[CFPTable], ValueQ[CFPAssoc]],Return[]];

  PrintTemporary["Loading CFPtable ..."];
  CFPTablefname = FileNameJoin[{moduleDir, "data", "CFPTable.m"}];
  If[!FileExistsQ[CFPTablefname],
    (PrintTemporary[">> CFPTable.m not found, generating ..."];
      CFPTable = GenerateCFPTable["Export"->True];
    ),
    CFPTable = Import[CFPTablefname];
  ];

  PrintTemporary["Loading CFPs.m ..."];
  CFPfname = FileNameJoin[{moduleDir, "data", "CFPs.m"}];
  If[!FileExistsQ[CFPfname],
    (PrintTemporary[">> CFPs.m not found, generating ..."];
      CFP = GenerateCFP["Export"->True];
    ),
    CFP = Import[CFPfname];
  ];

  PrintTemporary["Loading CFPAssoc.m ..."];
  CFPAfname = FileNameJoin[{moduleDir, "data", "CFPAssoc.m"}];
  If[!FileExistsQ[CFPAfname],
    (PrintTemporary[">> CFPAssoc.m not found, generating ..."];
      CFPAssoc = GenerateCFPAssoc["Export"->True];
    ),
    CFPAssoc = Import[CFPAfname];
  ];
);

ReducedUkTable::usage = "ReducedUkTable[{n, l = 3, SL, SpLp, k}]
 provides reduced matrix elements of the spherical tensor operator
 Uk. See TASS section 11-9 \"Unit Tensor Operators\". Loaded using
 LoadUk[].";

LoadUk::usage="LoadUk[] loads into session the reduced matrix
 elements for unit tensor operators.";
LoadUk[]:=(
  If[ValueQ[ReducedUkTable], Return[]];
  PrintTemporary["Loading the association of reduced matrix elements for unit tensor operators ..."];
  ReducedUkTableFname = FileNameJoin[{moduleDir, "data", "ReducedUkTable.m"}];
  If[!FileExistsQ[ReducedUkTableFname],
    (PrintTemporary[">> ReducedUkTable.m not found, generating ..."];
      ReducedUkTable = GenerateReducedUkTable[7];
    ),
    ReducedUkTable = Import[ReducedUkTableFname];
  ];
);

ElectrostaticTable::usage = "ElectrostaticTable[{n, SL, SpLp}] provides the calculated result of Electrostatic[{n, SL, SpLp}]. Load using LoadElectrostatic[].";

LoadElectrostatic::usage="LoadElectrostatic[] loads the reduced matrix elements for the electrostatic interaction.";
LoadElectrostatic[]:=(
  If[ValueQ[ElectrostaticTable], Return[]];
  PrintTemporary["Loading the association of matrix elements for the electrostatic interaction ..."];
  ElectrostaticTablefname = FileNameJoin[{moduleDir, "data", "ElectrostaticTable.m"}];
  If[!FileExistsQ[ElectrostaticTablefname],
    (PrintTemporary[">> ElectrostaticTable.m not found, generating ..."];
      ElectrostaticTable = GenerateElectrostaticTable[7];
    ),
    ElectrostaticTable = Import[ElectrostaticTablefname];
  ];
);

LoadV1k::usage="LoadV1k[] loads into session the matrix elements of V1k.";
LoadV1k[]:=(
  If[ValueQ[ReducedV1kTable], Return[]];
  PrintTemporary["Loading the association of matrix elements for V1k ..."];
  ReducedV1kTableFname = FileNameJoin[{moduleDir, "data", "ReducedV1kTable.m"}];
  If[!FileExistsQ[ReducedV1kTableFname],
    (PrintTemporary[">> ReducedV1kTable.m not found, generating ..."];
      ReducedV1kTable = GenerateReducedV1kTable[7];
    ),
    ReducedV1kTable = Import[ReducedV1kTableFname];
  ]
);

LoadSpinOrbit::usage="LoadSpinOrbit[] loads into session the matrix elements of the spin-orbit interaction.";
LoadSpinOrbit[]:=(
  If[ValueQ[SpinOrbitTable], Return[]];
  PrintTemporary["Loading the association of matrix elements for spin-orbit ..."];
  SpinOrbitTableFname = FileNameJoin[{moduleDir, "data", "SpinOrbitTable.m"}];
  If[!FileExistsQ[SpinOrbitTableFname],
    (PrintTemporary[">> SpinOrbitTable.m not found, generating ..."];
      SpinOrbitTable = GenerateSpinOrbitTable[7, "Export" -> True];
    ),
    SpinOrbitTable = Import[SpinOrbitTableFname];
  ]
);

LoadSOOandECSOLS::usage="LoadSOOandECSOLS[] loads into session the LS reduced matrix elements of the SOO-ECSO interaction.";
LoadSOOandECSOLS[]:=(
  If[ValueQ[SOOandECSOLSTable], Return[]];
  PrintTemporary["Loading the association of LS reduced matrix
  elements for SOO-ECSO ..."];
  SOOandECSOLSTableFname = FileNameJoin[{moduleDir, "data", "ReducedSOOandECSOLSTable.m"}];
  If[!FileExistsQ[SOOandECSOLSTableFname],
    (PrintTemporary[">> ReducedSOOandECSOLSTable.m not found, generating ..."];
      SOOandECSOLSTable = GenerateSOOandECSOLSTable[7];
    ),
    SOOandECSOLSTable = Import[SOOandECSOLSTableFname];
  ];
);

LoadSOOandECSO::usage="LoadSOOandECSO[] loads into session the LSJ reduced matrix elements of spin-other-orbit and electrostatically-correlated-spin-orbit.";
LoadSOOandECSO[]:=(
  If[ValueQ[SOOandECSOTableFname], Return[]];
  PrintTemporary["Loading the association of matrix elements for spin-other-orbit and electrostatically-correlated-spin-orbit ..."];
  SOOandECSOTableFname = FileNameJoin[{moduleDir, "data", "SOOandECSOTable.m"}];
  If[!FileExistsQ[SOOandECSOTableFname],
    (PrintTemporary[">> SOOandECSOTable.m not found, generating ..."];
      SOOandECSOTable = GenerateSOOandECSOTable[7, "Export"->True];
    ),
    SOOandECSOTable = Import[SOOandECSOTableFname];
  ];
);

LoadT22::usage="LoadT22[] loads into session the matrix elements of T22.";
LoadT22[]:=(
  If[ValueQ[T22Table], Return[]];
  PrintTemporary["Loading the association of reduced T22 matrix elements ..."];
  T22TableFname = FileNameJoin[{moduleDir, "data", "ReducedT22Table.m"}];
  If[!FileExistsQ[T22TableFname],
    (PrintTemporary[">> ReducedT22Table.m not found, generating ..."];
      T22Table = GenerateT22Table[7];
    ),
    T22Table = Import[T22TableFname];
  ];
);

LoadSpinSpin::usage="LoadSpinSpin[] loads into session the matrix elements of spin-spin.";
LoadSpinSpin[]:=(
  If[ValueQ[SpinSpinTable], Return[]];
  PrintTemporary["Loading the association of matrix elements for spin-spin ..."];
  SpinSpinTableFname = FileNameJoin[{moduleDir, "data", "SpinSpinTable.m"}];
  If[!FileExistsQ[SpinSpinTableFname],
    (PrintTemporary[">> SpinSpinTable.m not found, generating ..."];
      SpinSpinTable = GenerateSpinSpinTable[7, "Export" -> True];
    ),
    SpinSpinTable = Import[SpinSpinTableFname];
  ];
);

LoadThreeBody::usage="LoadThreeBody[] loads into session the matrix elements of three-body configuration-interaction effects.";
LoadThreeBody[]:=(
  If[ValueQ[ThreeBodyTable], Return[]];
  PrintTemporary["Loading the association of matrix elements for three-body configuration-interaction effects ..."];
  ThreeBodyFname   = FileNameJoin[{moduleDir, "data", "ThreeBodyTable.m"}];
  ThreeBodiesFname = FileNameJoin[{moduleDir, "data", "ThreeBodyTables.m"}];
```

```
    If[!FileExistsQ[ThreeBodyFname],
      (PrintTemporary[">> ThreeBodyTable.m not found, generating ..."
    ];
        {ThreeBodyTable, ThreeBodyTables} = GenerateThreeBodyTables
    [14, "Export" -> True];
      ),
      ThreeBodyTable = Import[ThreeBodyFname];
      ThreeBodyTables = Import[ThreeBodiesFname];
    ];
  );

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

End[]

LoadTermLabels[];
LoadCFP[];

EndPackage[]
```

5 qonstants.m

```
BeginPackage["qonstants`"];

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

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

(* SI *)
\[Mu]0  = 4 \[Pi]*10^-7;              (* magnetic permeability in
    vacuum in SI *)
hPlanck = 6.62607015*10^-34;          (* Planck's constant in SI *)
\[Mu]B  = 9.2740100783*10^-24;        (* Bohr magneton in SI *)
me      = 9.1093837015*10^-31;        (* electron mass in SI *)
cLight  = 2.99792458*10^8;            (* speed of light in SI *)
eCharge = 1.602176634*10^-19;         (* elementary charge in SI *)
αFine = 1/137.036;             (* fine structure constant in SI *)
bohrRadius   = 5.29177210903*10^-11; (* Bohr radius in SI *)

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

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

EndPackage[];
```

6 qplotter.m

```

BeginPackage["qplotter`"];

GetColor;
IndexMappingPlot;
ListLabelPlot;
AutoGraphicsGrid;
SpectrumPlot;
```

```
WaveToRGB;

Begin["`Private`"];

  AutoGraphicsGrid::usage="AutoGraphicsGrid[graphsList] takes a list of graphics and creates a GraphicsGrid with them. The number of columns and rows is chosen automatically so that the grid has a squarish shape.";
  Options[AutoGraphicsGrid] = Options[GraphicsGrid];
  AutoGraphicsGrid[graphsList_, opts : OptionsPattern[]] :=
    (
      numGraphs = Length[graphsList];
      width = Floor[Sqrt[numGraphs]];
      height = Ceiling[numGraphs/width];
      groupedGraphs = Partition[graphsList, width, width, 1, Null];
      GraphicsGrid[groupedGraphs, opts]
    )

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

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

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

  ListLabelPlot::usage="ListLabelPlot[data, labels] takes a list of numbers with corresponding labels. The data is grouped according
    to the labels and a ListPlot is created with them so that each
    group has a different color and their corresponding label is shown
     in the horizontal axis.";
  Options[ListLabelPlot] = Append[Options[ListPlot], "TickCompression"
    ->True];
  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[#, 0], 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,
      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 with to height ratio
  of the resulting graph is widthToHeightAspect. The color of the
  lines is determined by the wavelength of the line. The function
  assumes that the lines are given in nm.
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[];
```

7 misc.m

```
BeginPackage["misc`"];

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

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

FirstOrderPerturbation;
SecondOrderPerturbation;
RoundValueWithUncertainty;

ToPythonSymPyExpression;
RoundToSignificantFigures;
RobustMissingQ;

Begin["`Private`"];

  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 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 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. 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.";
FirstOrderPerturbation[eigenValues_, eigenVectors_,
  perMatrix_] := (Diagonal[
  eigenVectors . perMatrix . Transpose[eigenVectors]])

SecondOrderPerturbation::usage="Given the eigenValues and
  eigenVectors of a matrix A (which doesn't need to be given)
  together with a corresponding perturbation matrix perMatrix, this
  function calculates the second derivative of the eigenvalues with
  respect to the scale factor of the perturbation matrix. In the
  sense that the eigenvalues of the matrix A + β perMatrix are to
  second order equal to \[Lambda] + \[Delta]_i β + \[Delta]_i^{(2)
  }/2 β^2, where the \[Delta]_i^{(2)} are the returned values. The
  eigenvalues and eigenvectors are assumed to be given in the same
  order, i.e. the ith eigenvalue corresponds to the ith eigenvector.
   This assuming that the eigenvalues are non-degenerate.";
SecondOrderPerturbation[eigenValues_, eigenVectors_, perMatrix_] :=
  (
  dim = Length[perMatrix];
```

```
 eigenBras = Conjugate[eigenVectors];
 eigenKets = eigenVectors;
 matV = Abs[eigenBras . perMatrix . Transpose[eigenKets]]^2;
 OneOver[x_, y_] := If[x == y, 0, 1/(x - y)];
 eigenDiffs = Outer[OneOver, eigenValues, eigenValues, 1];
 pProduct = Transpose[eigenDiffs]*matV;
 Return[2*(Total /@ Transpose[pProduct])];
 )

SuperIdentity::usage="SuperIdentity[args] returns the arguments
 passed to it. This is useful for defining a function that does
 nothing, but that can be used in a composition.";
SuperIdentity[args___] := {args};

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

RecoverBasis::usage="RecoverBasis[{flatBasisLS, flatbasisNums}]
 takes the output of FlattenBasis and returns the original basis.
 The input is a list with two elements {flatbasisLS, flatbasisNums
 }.";
RecoverBasis[{flatbasisLS_, flatbasisNums_}] := Module[{recBasis},
 (
 recBasis = {{{#[[1]], #[[2]]}, #[[3]]}, #[[4]]} & /@ (Flatten /@
     Transpose[{flatbasisLS,
       Partition[Round[2*#]/2 & /@ flatbasisNums, 3]}]);
 Return[recBasis];
 )
 ]

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

ExportToH5::usage =
 "ExportToH5[fname, Hold[{symbol1, symbol2, ...}]] takes an .h5
 filename and a held list of symbols and export to the .h5 file the
  values of the symbols with keys equal the symbol names. The
 values of the symbols cannot be arbitrary, for instance a list
 with mixes numbers and string will fail, but an Association with
 mixed values exports ok. Do give it a try.
 If the file is already present in disk, this function will
 overwrite it by default. If the value of a given symbol contains
 symbolic numbers, e.g. \[Pi], these will be converted to floats in
  the exported file.";
Options[ExportToH5] = {"Overwrite" -> True};
ExportToH5[fname_String, symbols_Hold, OptionsPattern[]] := (
 If[And[FileExistsQ[fname], OptionValue["Overwrite"]],
 (
  Print["File already exists, overwriting ..."];
  DeleteFile[fname];
  )
 ];
 symbolNames = ExtractSymbolNames[symbols];
 Do[(Print[symbolName];
  Export[fname, ToExpression[symbolName], {"Datasets", symbolName
 },
  OverwriteTarget -> "Append"]
```

```
    ), {symbolName, symbolNames}]
  )

GreedyMatching::usage="GreedyMatching[aList, bList] returns a list
  of pairs of elements from aList and bList that are closest to each
   other, this is returned in a list together with a mapping of
  indices from the aList to those in bList to which they were
  matched. The option \"alistLabels\" can be used to specify labels
  for the elements in aList. The option \"blistLabels\" can be used
  to specify labels for the elements in bList. If these options are
  used, the function returns a list with three elements the pairs of
   matched elements, the pairs of corresponding matched labels, and
  the mapping of indices.";
Options[GreedyMatching] = {
    "alistLabels" -> {},
    "blistLabels" -> {}};
GreedyMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
  aValues = aValues0,
  bValues = bValues0,
  bValuesOriginal = bValues0,
  bestLabels, bestMatches,
  bestLabel, aElement, givenLabels,
  aLabels, aLabel,
  diffs, minDiff,
  bLabels,
  minDiffPosition, bestMatch},
  (
  aLabels     = OptionValue["alistLabels"];
  bLabels     = OptionValue["blistLabels"];
  bestMatches = {};
  bestLabels  = {};
  givenLabels = (Length[aLabels] > 0);
  Do[
    (
    aElement        = aValues[[idx]];
    diffs           = Abs[bValues - aElement];
    minDiff         = Min[diffs];
    minDiffPosition = Position[diffs, minDiff][[1, 1]];
    bestMatch       = bValues[[minDiffPosition]];
    bestMatches     = Append[bestMatches, {aElement, bestMatch}];
    If[givenLabels,
      (
      aLabel     = aLabels[[idx]];
      bestLabel  = bLabels[[minDiffPosition]];
      bestLabels = Append[bestLabels, {aLabel, bestLabel}];
      bLabels    = Drop[bLabels, {minDiffPosition}];
      )
      ];
    bValues = Drop[bValues, {minDiffPosition}];
    If[Length[bValues] == 0, Break[]];
    ),
    {idx, 1, Length[aValues]}
    ];
  pairedIndices = MapIndexed[{#2[[1]], Position[bValuesOriginal,
 #1[[2]]][[1, 1]]} &, bestMatches];
  If[givenLabels,
    Return[{bestMatches, bestLabels, pairedIndices}],
    Return[{bestMatches, pairedIndices}]
    ]
  )
  ]

StochasticMatching::usage="StochasticMatching[aValues, bValues]
  finds a better assignment by randomly shuffling the elements of
  aValues and then applying the greedy assignment algorithm. The
  function prints what is the range of total absolute differences
  found during shuffling, the standard deviation of all of them, and
   the number of shuffles that were attempted. The option \"
  alistLabels\" can be used to specify labels for the elements in
  aValues. The option \"blistLabels\" can be used to specify labels
  for the elements in bValues. If these options are used, the
  function returns a list with three elements the pairs of matched
  elements, the pairs of corresponding matched labels, and the
  mapping of indices.";
```

```
Options[StochasticMatching] = {"alistLabels" -> {},
  "blistLabels" -> {}};
StochasticMatching[aValues0_, bValues0_, numShuffles_ : 200,
 OptionsPattern[]] := Module[{
  aValues = aValues0,
  bValues = bValues0,
  matchingLabels, ranger, matches, noShuff, bestMatch, highestCost,
 lowestCost, dev, sorter, bestValues,
  pairedIndices, bestLabels, matchedIndices, shuffler
  },
  (
  matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
  ranger = Range[1, Length[aValues]];
  matches = If[Not[matchingLabels], (
      Table[(
        shuffler = If[i == 1, ranger, RandomSample[ranger]];
        {bestValues, matchedIndices} =
        GreedyMatching[aValues[[shuffler]], bValues];
        cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
        {cost, {bestValues, matchedIndices}}
        ), {i, 1, numShuffles}]
      ),
    Table[(
      shuffler = If[i == 1, ranger, RandomSample[ranger]];
      {bestValues, bestLabels, matchedIndices} =
        GreedyMatching[aValues[[shuffler]], bValues,
        "alistLabels" -> OptionValue["alistLabels"][[shuffler]],
        "blistLabels" -> OptionValue["blistLabels"]];
      cost = Total[Abs[#[[1]] - #[[2]]] & /@ bestValues];
      {cost, {bestValues, bestLabels, matchedIndices}}
      ), {i, 1, numShuffles}]
    ];
  noShuff = matches[[1, 1]];
  matches = SortBy[matches, First];
  bestMatch = matches[[1, 2]];
  highestCost = matches[[-1, 1]];
  lowestCost = matches[[1, 1]];
  dev = StandardDeviation[First /@ matches];
  Print[lowestCost, " <-> ", highestCost, " | \[Sigma]=", dev,
    " | N=", numShuffles, " | null=", noShuff];
  If[matchingLabels,
    (
    {bestValues, bestLabels, matchedIndices} = bestMatch;
    sorter = Ordering[First /@ bestValues];
    bestValues = bestValues[[sorter]];
    bestLabels = bestLabels[[sorter]];
    pairedIndices =
      MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
      bestValues];
    Return[{bestValues, bestLabels, pairedIndices}]
    ),
    (
    {bestValues, matchedIndices} = bestMatch;
    sorter = Ordering[First /@ bestValues];
    bestValues = bestValues[[sorter]];
    pairedIndices =
      MapIndexed[{#2[[1]], Position[bValues, #1[[2]]][[1, 1]]} &,
      bestValues];
    Return[{bestValues, pairedIndices}]
    )
    ];
  )
  ]

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
  \"noMatched\" 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 \"noMatched\" 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.";
Options[FlowMatching] = {"alistLabels" -> {}, "blistLabels" -> {}, "
  notMatched" -> 0, "CostFun"-> (Abs[#1-#2] &)};
FlowMatching[aValues0_, bValues0_, OptionsPattern[]] := Module[{
   aValues = aValues0, bValues = bValues0, edgesSourceToA,
  capacitySourceToA, nA, nB,
   costSourceToA, midLayer, midLayerEdges, midCapacities,
   midCosts, edgesBtoSink, capacityBtoSink, costBtoSink,
   allCapacities, allCosts, allEdges, graph,
   flow, bestValues, bestLabels, cFun,
   aLabels, bLabels, pairedIndices, matchingLabels},
   (
   matchingLabels = (Length[OptionValue["alistLabels"]] > 0);
   aLabels = OptionValue["alistLabels"];
   bLabels = OptionValue["blistLabels"];
   cFun = OptionValue["CostFun"];
   nA      = Length[aValues];
   nB      = Length[bValues];
   (*Build up the edges costs and capacities*)
   (*From source to the nodes representing the values of the first \
list*)
   edgesSourceToA = ("source" \[DirectedEdge] {"A", #}) & /@ Range[1,
    nA];
   capacitySourceToA = ConstantArray[1, nA];
   costSourceToA = ConstantArray[0, nA];

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

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

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

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

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

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

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

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

8 qalculations.m

```
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
    Hamiltonian are simply taken from the values quoted by Carnall. It
     uses precomputed symbolic matrices for the Hamiltonian so it's
    faster than the previous alternatives.

The function returns a list with nine elements
{rmsDifference, carnallEnergies, eigenEnergies, ln, carnallAssignments
    , simplerStateLabels, eigensys, basis, truncatedStates}.

Where:

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;

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. Same as eigensys but the eigenvectors have been truncated so that the truncated version adds up to at least a total probability of eigenstateTruncationProbability.

";
Options[FastIonSolverLaF3] = {
  "MakeNotebook" -> True,
  "NotebookSave" -> True,
  "HTMLSave" -> False,
  "eigenstateTruncationProbability" -> 0.9,
  "Include spin-spin" -> True,
  "Max Eigenstates in Table" -> 100,
  "Sparse" -> True,
  "PrintFun" -> Print,
  "SaveData" -> True,
  "paramFiddle" -> {},
  "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";
    paramFiddle = OptionValue["paramFiddle"];
    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;
    ];

    Do[params[key] = paramFiddle[key],
      {key, Keys[paramFiddle]}
    ];

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

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

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

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

(*Compare against the data quoted by Bill Carnall*)
PrintFun["> Comparing against the data from Carnall ..."];
mainKey            = StringTemplate["appendix:`Ln`:Association"][<|"Ln" -> ln|>];
lnData             = Carnall[mainKey];
carnalKeys         = lnData // Keys;
repetitions        = Length[lnData[#]["Calc (1/cm)"]] & /@ carnalKeys;
carnallAssignments = First /@ Carnall["appendix:" <> ln <> ":RawTable"];
carnalKey          = StringTemplate["appendix:`Ln`:Calculated"][<|"Ln" -> ln|>];
carnallEnergies    = Carnall[carnalKey];
```

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

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

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

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

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

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

```
    roundedDiffs = Round[diffs, 0.1];
    roundedDiffs = PadRight[roundedDiffs, Length[simplerStateLabels
  ], "-"];
    roundedDiffs = DefaultIfMissing /@ roundedDiffs;
    diffs = PadRight[diffs, Length[simplerStateLabels], "-"];
    diffs = DefaultIfMissing /@ diffs;
    diffTableData = Transpose[{simplerStateLabels, eigenEnergies,
      labelComparison,
      PadRight[carnallAssignments, Length[simplerStateLabels], "-"],
      DefaultIfMissing/@PadRight[carnallEnergies, Length[
  simplerStateLabels], "-"],
      roundedDiffs}
    ];
    diffTable = TableForm[diffTableData,
      TableHeadings -> {None, {"qlanth",
      "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)", "", "Carnall",
      "E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)",
      "\[CapitalDelta]E/\!\(\*SuperscriptBox[\(cm\), \(-1\)]\)"}}
    ];

    diffs = Sort[eigenEnergies][[;; Length[carnallEnergies]]] -
  carnallEnergies;
    notBad = FreeQ[#,Missing[]]&/@diffs;
    diffs = Pick[diffs,notBad];
    diffHistogram = Histogram[diffs,
      Frame -> True,
      ImageSize -> 800,
      AspectRatio -> 1/3, FrameStyle -> Directive[16],
      FrameLabel -> {"(qlanth-carnall)/Ky", "Freq"}
    ];
    rmsDifference = Sqrt[Total[diffs^2/Length[diffs]]];
    labelTempate = StringTemplate["\!\(\*SuperscriptBox[\(`ln`\),
  \(\(3\)\(+\)\)]\)"];
    diffData   = diffs;
    diffLabels = simplerStateLabels[[;;Length[notBad]]];
    diffLabels = Pick[diffLabels, notBad];
    diffPlot   = Framed[
      ListLabelPlot[diffData,
      diffLabels,
      Frame -> True,
      PlotRange -> All,
      ImageSize -> 1200,
      AspectRatio -> 1/3,
      FrameLabel -> {"",
      "(qlanth-carnall) / \!\(\*SuperscriptBox[\(cm\), \(-1\)]\)"},
      PlotMarkers -> "OpenMarkers",
      PlotLabel ->
      Style[labelTempate[<|"ln" -> ln|>] <> " | " <> "\[Sigma]=" <>
          ToString[Round[rmsDifference, 0.01]] <>
          " \!\(\*SuperscriptBox[\(cm\), \(-1\)]\)\n", 20],
      Background -> White
      ],
      Background -> White,
      FrameMargins -> 50
    ];
    (* now place all of this in a new notebook *)
    nb = CreateDocument[
    {
      TextCell[Style[
        DisplayForm[RowBox[{SuperscriptBox[host <> ":" <> ln, "3+"],
   "(", SuperscriptBox["f", numE], ")"}]]
        ], "Title", TextAlignment -> Center
      ],
      TextCell["Energy Diagram",
        "Section",
        TextAlignment -> Center
      ],
      TextCell[energyDiagram,
        TextAlignment -> Center
      ],
      TextCell["Multiplet Assignments & Energy Levels",
        "Section",
        TextAlignment -> Center
      ],
      TextCell[diffHistogram, TextAlignment -> Center],
      TextCell[diffPlot, "Output", TextAlignment -> Center],
      TextCell[assignmentMatches, "Output", TextAlignment -> Center
  ],
      TextCell[diffTable, "Output", TextAlignment -> Center],
      TextCell["Truncated Eigenstates", "Section", TextAlignment ->
  Center],
      TextCell["These are some of the resultant eigenstates which
  add up to at least a total probability of " <> ToString[
  eigenstateTruncationProbability] <> ".", "Text", TextAlignment ->
  Center],
      TextCell[statesTable, "Output", TextAlignment -> Center]
    },
    WindowSelected -> True,
    WindowTitle -> ln <> " in " <> "LaF3" <> appToFname,
    WindowSize -> {1600, 800}];
    If[OptionValue["SaveData"],
    (
      exportFname = FileNameJoin[{workDir,OptionValue["
  OutputDirectory"], ln <> " in " <> "LaF3" <> appToFname <> ".m"}];
      SelectionMove[nb, After, Notebook];
      NotebookWrite[nb, Cell["Reload Data", "Section", TextAlignment
   -> Center]];
      NotebookWrite[nb,
        Cell[(
          "{rmsDifference, carnallEnergies, eigenEnergies, ln,
  carnallAssignments, simplerStateLabels, eigensys, basis,
  truncatedStates} = Import[FileNameJoin[{NotebookDirectory[],\"" <>
   StringSplit[exportFname,"/"][[-1]] <> "\"}]];"
          ),"Input"
        ]
      ];
      NotebookWrite[nb,
        Cell[(
          "Manipulate[First[MinimalBy[truncatedStates, Abs[First[#]
  - energy] &]], {energy,0}]"
          ),"Input"]
      ];
      (* Move the cursor to the top of the notebook *)
      SelectionMove[nb, Before, Notebook];
      Export[exportFname,
        {rmsDifference, carnallEnergies, eigenEnergies, ln,
  carnallAssignments, simplerStateLabels, eigensys, basis,
  truncatedStates}
      ];
      tinyexportFname = FileNameJoin[
        {workDir, OptionValue["OutputDirectory"], ln <> " in " <> "
  LaF3" <> appToFname <> " - tiny.m"}
      ];
      tinyExport = <|"ln"->ln,
                      "carnallEnergies"->carnallEnergies,
                      "rmsDifference"-> rmsDifference,
                      "eigenEnergies"-> eigenEnergies,
                      "carnallAssignments"-> carnallAssignments,
                      "simplerStateLabels" -> simplerStateLabels|>;
      Export[tinyexportFname, tinyExport];
    )
    ];
    If[OptionValue["NotebookSave"],
      (
        nbFname = FileNameJoin[{workDir,OptionValue["OutputDirectory
  "], ln <> " in " <> "LaF3" <> appToFname <> ".nb"}];
        PrintFun[">> Saving notebook to ", nbFname, " ..."];
        NotebookSave[nb, nbFname];
      )
    ];
    If[OptionValue["HTMLSave"],
    (
      htmlFname = FileNameJoin[{workDir,OptionValue["OutputDirectory
  "], "html", ln <> " in " <> "LaF3" <> appToFname <> ".html"}];
      PrintFun[">> Saving html version to ", htmlFname, " ..."];
      Export[htmlFname, nb];
    )
    ];
```

```
    )
    ];

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

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

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

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

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

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

The function returns an association containing the following keys:
    Line Strength, AMD, fMD, Radiative lifetimes, Transition Energies
    / K, Transition Wavelengths in nm.";
Options[MagneticDipoleTransitions] = {
        "Make Notebook" -> True,
        "Close Notebook" -> True,
        "Print Function" -> PrintTemporary,
        "Host" -> "LaF3",
        "Wavelength Range" -> {50,2000}};
MagneticDipoleTransitions[numE_Integer, OptionsPattern[]]:= (
  host           = OptionValue["Host"];
  \[Lambda]Range = OptionValue["Wavelength Range"];
  PrintFun       = OptionValue["Print Function"];
  {\[Lambda]min, \[Lambda]max} = OptionValue["Wavelength Range"];

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

```

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

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

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

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

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

PrintFun["Calculating the transition energies in K ..."];
transitionEnergies=Outer[Subtract,First/@eigensys,First/@eigensys];
magIon["Transition Energies / K"]=ReplaceDiagonal[transitionEnergies,Indeterminate];

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

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

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

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

```
  [[8]]}]];
allTransitions[[7]]          = roundedWaves;
allTransitions[[8]]          = roundedDeltas;

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

(* 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"];

If[OptionValue["Make Notebook"],
(
    PrintFun["Creating a notebook with a Manipulate to select a wavelength interval and a lifetime power of ten ..."];
    finalTable     = Rest[finalTable];
    finalTable     = SortBy[finalTable,#[[7]]&];
    opticalTable   = Select[finalTable,\[Lambda]min<=#[[7]]<=\[Lambda]max&];
    pows           = Sort[DeleteDuplicates[(MantissaExponent[#[[9]]][[2]]-1)&/@opticalTable]];

    man            = Manipulate[
    (
        {\[Lambda]min,\[Lambda]max} = \[Lambda]int;
        table = Select[opticalTable,And[(\[Lambda]min<=#[[7]]<=\[Lambda]max),
                        (MantissaExponent[#[[9]]][[2]]-1)==log10\[Tau]]&];
        tab   = TableForm[table,TableHeadings->{None,header}];
        Column[{{"\[Lambda]min="<>ToString[\[Lambda]min]<>" nm","\[Lambda]max="<>ToString[\[Lambda]max]<>" nm",log10\[Tau]},tab}]
    ),
    {{\[Lambda]int,\[Lambda]Range,"\[Lambda] interval"},
        \[Lambda]Range[[1]],
        \[Lambda]Range[[2]],
        50,
        ControlType->IntervalSlider
    },
    {{log10\[Tau],pows[[-1]]},
        pows
    },
    TrackedSymbols  :> {\[Lambda]int,log10\[Tau]},
        SaveDefinitions -> True
        ];

        nb = CreateDocument[{
            TextCell[Style[DisplayForm[RowBox[{"Magnetic Dipole
    Transitions", "\n", SuperscriptBox[host<>":"<>ln,"3+"],"(",
    SuperscriptBox["f",numE],")"}]]],"Title",TextAlignment->Center],
            (* TextCell["Magnetic Dipole Transition Lifetimes","Section
    ",TextAlignment->Center], *)
            TextCell[man,"Output",TextAlignment->Center]
        },
        WindowSelected -> True,
        WindowTitle    -> "MD1 - "<>ln<>" in "<>host,
        WindowSize     -> {1600,800}
        ];
        SelectionMove[nb, After, Notebook];
        NotebookWrite[nb, Cell["Reload Data", "Section", TextAlignment
    -> Center]];
        NotebookWrite[nb, Cell[(
              "magTransitions = Import[FileNameJoin[{NotebookDirectory
    [],\"" <> StringSplit[rawexportFname,"/"][[-1]] <> "\"}],\""<>
    rawexportKey<>"\"];"
              ),"Input"]];
        SelectionMove[nb, Before, Notebook];
        nbFname = FileNameJoin[{workDir,"examples","MD1 - "<>ln<>" in "
    <>"LaF3"<>".nb"}];
        PrintFun[">> Saving notebook to ",nbFname," ..."];
        NotebookSave[nb, nbFname];
        If[OptionValue["Close Notebook"],
            NotebookClose[nb];
        ];
    )
    ];

    Return[magIon];
)
```

References

[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).

[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.

[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.

[MR71] JC Morrison and K Rajnak. "Many-body calculations for the heavy atoms". In: *Physical Review A* 4.2 (1971). Publisher: APS, p. 536.