```
/*
 *  Basic core of a program to calculate the energies of the transitions
 *  of a quadrupolar nucleus. This program is an implementation and
 *  illustration of the methods in the paper in Molecular Physics, vol 101
 *  pp 3163-3173 (2003)
 *
 *  The program is distributed under the terms of the GNU public licence
 *
 *  Copyright (c) by Alex D. Bain, 2004
 */
#include <math.h>
#include <stdio.h>
#include "zmatrix2.h"  /* collection of meschach headers */

#define VERSION "040327.1"

#define SQR(x) ((x) * (x))

#define PI 3.14159265358979323846
#define NPOINTS (8*256) /* 65536 for 64K*/
#define SCALE 1000.0
#define ANGLE_ERROR 0.0  /* magic = 0.955316618124509 radians*/

typedef struct
  {
    short superspin;           /* characterized by total angular momentum */
    short zcomponent;          /* and its z component                     */
  }    basis_element;

typedef struct
  {
    double nu_zero;         /* Larmor freq */
    double e2qQ;            /* quadrupole interaction */
    double eta;             /* asymmetry */
    double alpha;           /* orientation of quadrupole */
    double beta;
    double gamma;
    double sigma11;         /* shift tensor elements */
    double sigma22;
    double sigma33;
    double phi;             /* orientation of shift tensor */
    double chi;
    double psi;
    double zeta;            /* angle of detector */
    double startf;
    double stopf;
    double relax;
    double O1;
    double omega1;          /* frequency of pulse */
    double gamB1;           /* gamma B1 for pulse */
    double pw;              /* pulse width */
  }    spin_params;

/* function prototypes */
void gen_spectrum(int spinX2, double theta, double phi, int n_sites,
               VEC *cryst_alph,VEC *cryst_beta, VEC *spectrum, ZVEC *transprob,
               ZVEC *rhozero, ZVEC *detector,
               ZMAT *L, spin_params *spinp, basis_element *basis_set);
void gen_Lvn(int spinX2, double theta, double phi, double shielding,
               ZMAT *L, spin_params *spinp, basis_element *basis_set);
double gen_shielding(double theta, double phi, spin_params *spinp);
void gen_qfreq(int spinX2, MAT *Lmatxr, MAT *Lmatxi, ZMAT *L, ZMAT *U,
                       VEC *eigvalsr, VEC *eigvalsi);
void mult_d2(double cbt, double sbt, ZVEC *in, ZVEC *out);
void mult_d1(double cbt, double sbt, ZVEC *in, ZVEC *out);
void mult_rz(double cal, double sal, ZVEC *in, ZVEC *out);
void define_qbasis(basis_element *basis_set, int spinX2);
int matrix_element_7ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set);
int matrix_element_5ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set);
```

```
int matrix_element_3ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set);
int matrix_element_1(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set);
void norm_1(ZMAT *U);
double threeJ_a_ap1_2(int a, int az, int cz);
void cg_(int nm, int n, double **ar, double **ai, double *wr, double *wi,
                int matz, double **zr, double **eigveci, double *fv1, double *fv2,
                double *fv3, int *ierr);

/*****************************************************************/
/********************* START OF MAIN PROGRAM **********************/
/*****************************************************************/
int main(int argc,char **argv)

{
   int i, j, k;
   int fibon_order, n_angles;
   int nsites;
   int spinX2, n_basis, absval;
   int n_per_cell;

   double theta, phi, freq, weight;
   double domega, fac, f0, f1;

   char fullfname[80];

   /*
    *  These are all data structures for the meschach package of
    *  numerical subroutines. The vector and matrix elements are
    *  not accessed directly, as in L[i][j], but as memebers of
    *  a structure L->me[i][j]. This is a hassle to begin with,
    *  but you get used to it.
    */

   ZMAT *L;
   VEC *spectrum;
   ZVEC *transprob;
   VEC *cryst_alph, *cryst_beta, *cryst_gamm;
   ZVEC *cspect;

   ZVEC *rhozero, *rhozerop, *Uinv_rho, *detector;
   PERM *eigorder;

   basis_element *basis_set;
   spin_params spinp;

   FILE *inpt;

   /*
    *  Start of the program
    */
   printf("Quadrupole program %s \n",VERSION);

   /* open the input file */
    //strcpy(fullfname, argv[1]);
    strcpy(fullfname, "inpt32.txt");
    if ((inpt=fopen(fullfname,"rt")) == NULL)
      {
        printf("cannot open input file.\n");
        exit(0);
      }

    /*
     *  Do all the input
     *  skipjunk is a meschach routine that skips blank lines and
     *  those that start with #
     *  This is useful, since it allows comments in the input file
     */
    skipjunk(inpt);
    fscanf(inpt, "%d", &spinX2);       /* spin quantum number */
```

```
skipjunk(inpt);
fscanf(inpt, "%d", &fibon_order);  /* fibonacci order, for ZCW powder average */
skipjunk(inpt);
fscanf(inpt, "%s", fullfname);     /* output file name (not used */
skipjunk(inpt);                    /* parameters for plot */
fscanf(inpt, "%lf %lf %lf", &spinp.startf, &spinp.stopf, &spinp.relax);
skipjunk(inpt);                    /* Larmor frequency, e^2qQ, and eta */
fscanf(inpt, "%lf %lf %lf", &spinp.nu_zero, &spinp.e2qQ, &spinp.eta);
spinp.nu_zero *= 2*PI;
spinp.e2qQ *= 2*PI;
skipjunk(inpt);                    /* orientation of the efg tensor */
fscanf(inpt, "%lf %lf  %lf", &spinp.alpha, &spinp.beta, &spinp.gamma);
spinp.alpha *= (PI/180.0);
spinp.beta *= (PI/180.0);
spinp.gamma *= (PI/180.0);
skipjunk(inpt);                    /* shift tensor elements */
fscanf(inpt, "%lf %lf  %lf", &spinp.sigma11, &spinp.sigma22, &spinp.sigma33);
skipjunk(inpt);                    /* orientation of shift tensor */
fscanf(inpt, "%lf %lf  %lf", &spinp.phi, &spinp.chi, &spinp.psi);
spinp.phi *= (PI/180.0);
spinp.chi *= (PI/180.0);
spinp.psi *= (PI/180.0);
skipjunk(inpt);                    /* angle of the crystal (if no ZCW average) */
fscanf(inpt, "%lf %lf ", &theta, &phi);
theta *= (PI/180.0);
phi *= (PI/180.0);

n_per_cell = 1;

/*
 *  The various get routines are meschach's memory allocation
 */
cryst_alph = v_get(n_per_cell);
cryst_beta = v_get(n_per_cell);
cryst_gamm = v_get(n_per_cell);

/* do the memory allocation for basis */
n_basis = SQR(spinX2 + 1) - 1;
if ((basis_set=(basis_element *)
   malloc(n_basis*sizeof(basis_element))) == NULL)
   exit(1);

define_qbasis(basis_set, spinX2);

/*
 * Set up the parameters - first, some memory allocation
 */

L = zm_get(n_basis, n_basis);
spectrum = v_get(n_basis);
transprob = zv_get(n_basis);

rhozero = zv_get(n_basis);
rhozerop = zv_get(n_basis);
Uinv_rho = zv_get(n_basis);
detector = zv_get(n_basis);

cspect = zv_get(NPOINTS);
zv_zero(cspect);

/*
 *  calculate the powder average over the whole sphere
 *  use ZCW grid, copied from the SIMPSON program
 */

nsites = 1;
cryst_alph->ve[0] = spinp.alpha;
cryst_alph->ve[0] = spinp.beta;
cryst_alph->ve[0] = 0.0;

n_angles = 1;
```

```
    /* In general, do a ZCW powder average over all angles */
    for (i=0; i<n_angles; i++)
      {
        weight = 1.0;
        gen_spectrum(spinX2, theta, phi, nsites, cryst_alph, cryst_beta,
                         spectrum, transprob, rhozero, detector, L, &spinp, basis_set);
      }

    /* list the frequencies */
    for (i=0; i<n_basis; i++)
      {
        printf(" %14.8f\n",spectrum->ve[i]);
      }

}   /* end of main program */

/*********************************************************************/

void gen_spectrum(int spinX2, double theta, double phi, int n_sites,
               VEC *cryst_alph,VEC *cryst_beta, VEC *spectrum, ZVEC *transprob,
               ZVEC *rhozero, ZVEC *detector,
               ZMAT *L, spin_params *spinp, basis_element *basis_set)
/*
 *  This is a driver program, which allows for multiple orientations within
 *  a single crystal, as in NaClO3
 */

{
    int n_basis, nlines;
    int i, j, k;

    double shielding;

    /* More meschach memory allocation tricks for temporary storage */
    static VEC *eigvalsr = VNULL;
    static VEC *eigvalsi = VNULL;
    static VEC *spectrum_unsort = VNULL;
    static VEC *spectrum_sort = VNULL;
    static ZVEC *trans_unsort;
    static ZVEC *rhoflip;
    static PERM *sort_order = PNULL;

    static MAT *Lmatxr = MNULL;
    static MAT *Lmatxi = MNULL;
    static ZMAT *U = ZMNULL;

    n_basis = SQR(spinX2 + 1) - 1;
    nlines = n_sites*(spinX2);

    eigvalsr = v_resize(eigvalsr, n_basis);
    eigvalsi = v_resize(eigvalsi, n_basis);
    Lmatxr = m_resize(Lmatxr, n_basis, n_basis);
    Lmatxi = m_resize(Lmatxi, n_basis, n_basis);
    U = zm_resize(U, n_basis, n_basis);
    spectrum_unsort = v_resize(spectrum_unsort, n_sites*n_basis);
    spectrum_sort = v_resize(spectrum_sort, n_sites*n_basis);
    trans_unsort = zv_resize(trans_unsort, n_sites*n_basis);
    rhoflip = zv_resize(rhoflip, n_basis);
    sort_order = px_resize(sort_order, n_sites*n_basis);

    /* Get started */
    k = 0;         /* counter for the transitions */
    for (i=0; i<n_sites; i++)
      {
        zm_zero(L);

        /* calculate the spectrum for a single nucleus */
        shielding = gen_shielding(theta, phi, spinp);
        gen_Lvn(spinX2, theta, phi, shielding, L, spinp, basis_set);
        gen_qfreq(spinX2, Lmatxr, Lmatxi, L, U, eigvalsr, eigvalsi);
```

```
        /*
         *  I don't quite trust the transition probability calculation yet,
         *  so it has been cut
         */

        /* accumulate the spectrum */
        for (j=0; j<n_basis; j++)
          {
            spectrum_unsort->ve[k] = eigvalsr->ve[j]/(2.0*PI);
            trans_unsort->ve[k].re = transprob->ve[j].re;
            trans_unsort->ve[k].im = transprob->ve[j].im;
            k++;
          }
      }   /* scanning over angles */

    spectrum_sort = v_sort(spectrum_unsort, sort_order);

    for(i=0; i<(n_sites*n_basis); i++)
      {
        spectrum->ve[i] = spectrum_sort->ve[i];
        transprob->ve[i].re = trans_unsort->ve[sort_order->pe[i]].re;
        transprob->ve[i].im = trans_unsort->ve[sort_order->pe[i]].im;
      }

}

/*****************************************************************/

void gen_Lvn(int spinX2, double theta, double phi, double shielding,
               ZMAT *L, spin_params *spinp, basis_element *basis_set)
/*
 *  Calculate and diagonalize the Liouvillian
 *  The spin parameters are all in that data structure
 *  There are two sets of angles - theta and phi, which are the orientation
 *  of the crystal with respect to the magnetic field, then alpha, beta
 *  and gammma (in spinp) which are the Euler angles of the tensor w.r.t.
 *  the crystal axes
 */
{
    int i, j, n_basis;
    double Qcoup;
    double cosa, sina;
    double rel, img;

    static ZVEC *Qplr = ZVNULL;
    static ZVEC *trQplr = ZVNULL;
    static ZVEC *wksp = ZVNULL;

    n_basis = SQR(spinX2+1) - 1;

    /*
     *  spatial part of the quadrupole Liouvillian,
     *  in principal axes fram, and transformed
     */
    Qplr = zv_resize(Qplr, 5);
    trQplr = zv_resize(trQplr, 5);

    /*
     *  Put in Zeeman interaction - we do all this in the
     *  lab frame, so Zeeman is just Iz
     */
    for (i=0; i<n_basis; i++)
        L->me[i][i].re = shielding * basis_set[i].zcomponent;

    /*
     *  Define the quadrupolar interaction as a spherical tensor
     *  in its principal axis frame
     */
    Qcoup = spinp->e2qQ/(spinX2*(spinX2-1)); /* convert e2qQ */

    zv_zero(Qplr);    /* equation 28 in the paper */
```

```
    /* use the 2+n index to remind us of spherical tensor components */
    Qplr->ve[2+2].re = -Qcoup * spinp->eta/2.0;
    Qplr->ve[2+0].re =  Qcoup * sqrt(3.0/2.0);
    Qplr->ve[2-2].re = -Qcoup * spinp->eta/2.0;

    /*
     *  Now rotate quadrupole into lab frame
     *  First, from PAS to crystal orientation,
     *  then from crystal to lab.
     *  Only two angles needed to orient the molecule
     *  in the magnetic field
     */
    wksp = zv_resize(wksp, 5);
    /* rotate the tensor first */
    cosa = cos(spinp->gamma);          sina = sin(spinp->gamma);
    mult_rz(cosa, sina, Qplr, trQplr);
    cosa = cos(spinp->beta);           sina = sin(spinp->beta);
    mult_d2(cosa, sina, trQplr, wksp);
    cosa = cos(spinp->alpha);          sina = sin(spinp->alpha);
    mult_rz(cosa, sina, wksp, trQplr);
    /* then the Euler angles of the crystal  */
    cosa = cos(phi);           sina = sin(phi);
    mult_rz(cosa, sina, trQplr, wksp);
    cosa = cos(theta);         sina = sin(theta);
    mult_d2(cosa, sina, wksp, trQplr);

    /* calculate the matrix elements of the Liouvillian */
    for (i=0; i<n_basis; i++)
      {
        for (j=i; j<n_basis; j++)
          {
            switch (spinX2)
               {  /* generate matrix element */
                 case 2 : matrix_element_1(i, j, &rel, &img, trQplr, basis_set);    /* spin 1 */
                   break;
                 case 3 : matrix_element_3ov2(i, j, &rel, &img, trQplr, basis_set); /* 3/2 */
                   break;
                 case 5 : matrix_element_5ov2(i, j, &rel, &img, trQplr, basis_set); /* 5/2 */
                   break;
                 case 7 : matrix_element_7ov2(i, j, &rel, &img, trQplr, basis_set); /* 7/2 */
                   break;
               }
            /* enter it into Liouvillian */
            L->me[i][j].re += rel;
            L->me[i][j].im -= img;
          }
      }

    /* make Hermitian */
    for (i=1; i<n_basis; i++)
      {
        for (j=0; j<i; j++)
          {
            L->me[i][j].re =  L->me[j][i].re;
            L->me[i][j].im = -L->me[j][i].im;
          }
      }

}

/*****************************************************************/

double gen_shielding(double theta, double phi, spin_params *spinp)
/*
 *  Calculate the chemical shilding from the tensor and
 *  the crystal orientation
 */
{
   double sigmaav, shielding;
   double cosa, sina;
```

```
   static ZVEC *sigma = ZVNULL;
   static ZVEC *trsig = ZVNULL;
   static ZVEC *wksp = ZVNULL;

    /* allocate temporary storage */
    sigma = zv_resize(sigma, 5);
    trsig = zv_resize(trsig, 5);
    wksp = zv_resize(wksp, 5);

    zv_zero(sigma);

    /* convert the shift tensor into a proper second-order spherical tensor */
    sigmaav = (spinp->sigma11+spinp->sigma22+spinp->sigma33)/3.0;

    /* use the subscripts to remind us of spherical tensor elements */
    sigma->ve[2+0].re = sqrt(1.0/6.0) *
                    (2*spinp->sigma33 - spinp->sigma11 - spinp->sigma22);
    sigma->ve[2+2].re = (1.0/2.0) * (spinp->sigma11 - spinp->sigma22);
    sigma->ve[2-2].re = (1.0/2.0) * (spinp->sigma11 - spinp->sigma22);

    /*
     *  Now rotate quadrupole into lab frame
     *  First, from PAS to crystal orientation,
     *  then from crystal to lab.
     *  Only two angles needed to orient the molecule
     *  in the magnetic field
     */
    wksp = zv_resize(wksp, 5);
    /* rotate the tensor first */
    cosa = cos(spinp->psi);          sina = sin(spinp->psi);
    mult_rz(cosa, sina, sigma, trsig);
    cosa = cos(spinp->chi);           sina = sin(spinp->chi);
    mult_d2(cosa, sina, trsig, wksp);
    cosa = cos(spinp->phi);          sina = sin(spinp->phi);
    mult_rz(cosa, sina, wksp, trsig);
    /* then the Euler angles of the crystal */
    cosa = cos(phi);           sina = sin(phi);
    mult_rz(cosa, sina, trsig, wksp);
    cosa = cos(theta);         sina = sin(theta);
    mult_d2(cosa, sina, wksp, trsig);

    shielding = spinp->nu_zero*(1.0 +
                   (sigmaav + 2*(1.0/sqrt(6.0))*trsig->ve[2+0].re)/1e6);
                   /* factor of 2 comes from remaining part of (2BzIz -B+I- -B-I+) */
    return(shielding);

}

/*****************************************************************/

void gen_qfreq(int spinX2, MAT *Lmatxr, MAT *Lmatxi, ZMAT *L, ZMAT *U,
                        VEC *eigvalsr, VEC *eigvalsi)
/*
 *  Interface to eigenvalue routine. This is an f2c translation of the
 *  EISPACK complex general routine. It's overkill, since the matrix is
 *  Hermitian, but it was left over from MEXICO and it works
 */
{
    int i, j, n_basis, matz, ierr;

    static VEC *fv1 = VNULL, *fv2 = VNULL, *fv3 = VNULL;
    static MAT *eigvecr = MNULL;
    static MAT *eigveci = MNULL;

    n_basis = SQR(spinX2+1) - 1;

    /* copy from a ZMAT, for eigenvalue routine */
    for (i=0; i<n_basis; i++)
      {
        for (j=0; j<n_basis; j++)
          {
```

```
            Lmatxr->me[i][j] = L->me[i][j].re;
            Lmatxi->me[i][j] = L->me[i][j].im;
          }
      }

    eigvecr = m_resize(eigvecr, n_basis, n_basis);
    eigveci = m_resize(eigveci, n_basis, n_basis);
    fv1 = v_resize(fv1, n_basis);
    fv2 = v_resize(fv2, n_basis);
    fv3 = v_resize(fv3, n_basis);

    /* diagonalize */
    matz = 1;
    cg_(n_basis, n_basis, Lmatxr->me, Lmatxi->me,
            eigvalsr->ve, eigvalsi->ve, matz, eigvecr->me, eigveci->me, fv1->ve, fv2->ve, fv3->ve,
&ierr);

    /*
     *  copy the eigenvectors back into a ZMAT and normalize them to a 1 norm
     *  biggest entry = 1 + 0i
     */

    for (i=0; i<n_basis; i++)
      {
        for (j=0; j<n_basis; j++)
            U->me[j][i] = zmake(eigvecr->me[j][i], eigveci->me[j][i]);
      }

    norm_1(U);  /* normalize to 1 norm */
}


/*********************************************************************/

int matrix_element_1(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set)
/*
 *  calculate matrix element using the Wigner-Eckart theorem for element ii, jj
 *  assume ii<jj, so superspins increase with basis set, as do z components
 *  the matrix elements ar of the form (Ii | 2k | Jj)
 *  according to Wigner = (-1)^(I-i) ( I 2 J) (I || 2 || J)
 *                                    (-i k j)
 *  we convert 3j symbol to (I J  2) with 2 minus signs, since J = I+1
 *                          (i j -k)
 *  otherwise matrix element is zero
 *  also include tensor contraction with rotated quadrupolar tensor, trQplr
 *  since this is the matrix element of operator with z component +k, we
 *  need the (-k) element of trQplr
 */

{
   int II, i, J, k;
   double x, y;

   II = basis_set[ii].superspin;
   i = basis_set[ii].zcomponent;
   J = basis_set[jj].superspin;

   /* quadrupole changes superspin by exactly 1 */
   if ((J - II) != 1)
     {
       *realp = 0.0;
       *imagp = 0.0;
       return(1);
     }

   /* second order tensor - change in z can't be > 2 */
   if ((abs(basis_set[ii].zcomponent - basis_set[jj].zcomponent)) > 2)
     {
       *realp = 0.0;
       *imagp = 0.0;
```

```
      return(1);
    }

  if (II == 1)
    {
      k = basis_set[ii].zcomponent - basis_set[jj].zcomponent;
      x = threeJ_a_ap1_2(II, i, -k) * (-sqrt(15.0));
      if ((II-i)%2) x = -x;          /* set sign */
      y = x * trQplr->ve[2-k].re;    /* include contraction */
      if (k%2) y = -y;               /* minus sign for contraction */
      *realp = y;                    /* return the value */
      y = x * trQplr->ve[2-k].im;
      if (k%2) y = -y;
      *imagp = y;
      return(1);
    }
}

/*********************************************************************/

int matrix_element_3ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set)
/*
 *  calculate matrix element using the Wigner-Eckart theorem for element ii, jj
 *  assume ii<jj, so superspins increase with basis set, as do z components
 *  the matrix elements ar of the form (Ii | 2k | Jj)
 *  according to Wigner = (-1)^(I-i) ( I 2 J) (I || 2 || J)
 *                                   (-i k j)
 *  we convert 3j symbol to (I J  2) with 2 minus signs, since J = I+1
 *                          (i j -k)
 *  otherwise matrix element is zero
 *  also include tensor contraction with rotated quadrupolar tensor, trQplr
 *  since this is the matrix element of operator with z component +k, we
 *  need the (-k) element of trQplr
 *  use II rather than I, since I is reserved in RedHat
 */

{
  int II, i, J, k;
  double x, y;

  II = basis_set[ii].superspin;
  i = basis_set[ii].zcomponent;
  J = basis_set[jj].superspin;

  /* quadrupole changes superspin by exactly 1 */
  if ((J - II) != 1)
    {
      *realp = 0.0;
      *imagp = 0.0;
      return(1);
    }

  /* second order tensor - change in z can't be > 2 */
  if ((abs(basis_set[ii].zcomponent - basis_set[jj].zcomponent)) > 2)
    {
      *realp = 0.0;
      *imagp = 0.0;
      return(1);
    }

  /*
   *  OK, we have a non-zero value
   *  This is equation 20 in the paper
   */
  k = basis_set[ii].zcomponent - basis_set[jj].zcomponent;
  x = threeJ_a_ap1_2(II, i, -k);  /* calculate the 3j symbol */
  switch (II)
    {  /* multiply by reduced matrix element */
      case 1 : x *= (-6.0);            /* (2 ||Q|| 1) */
               break;
```

```
       case 2 : x *= (-2*sqrt(21.0));   /* (3 ||Q|| 2) */
                break;
     }
   /* calculate (n modulo 2) as a way of setting (-1)^n in these calculations */
   if ((II-i)%2) x = -x;          /* set sign in Wigner Eckardt */
   y = x * trQplr->ve[2-k].re;    /* include contraction, k combines with -k */
   if (k%2) y = -y;               /* minus sign for tensor contraction */
   *realp = y;                    /* return the value */
   y = x * trQplr->ve[2-k].im;
   if (k%2) y = -y;
   *imagp = y;
   return(1);

}

/*****************************************************************/

int matrix_element_5ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set)
/*
 *  calculate matrix element using the Wigner-Eckart theorem for element ii, jj
 *  assume ii<jj, so superspins increase with basis set, as do z components
 *  the matrix elements ar of the form (Ii | 2k | Jj)
 *  according to Wigner = (-1)^(I-i) ( I 2 J) (I || 2 || J)
 *                                   (-i k j)
 *  we convert 3j symbol to (I J  2) with 2 minus signs, since J = I+1
 *                          (i j -k)
 *  otherwise matrix element is zero
 *  also include tensor contraction with rotated quadrupolar tensor, trQplr
 *  since this is the matrix element of operator with z component +k, we
 *  need the (-k) element of trQplr
 */

{
   int II, i, J, k;
   double x, y;

   II = basis_set[ii].superspin;
   i = basis_set[ii].zcomponent;
   J = basis_set[jj].superspin;

   /* quadrupole changes superspin by exactly 1 */
   if ((J - II) != 1)
     {
       *realp = 0.0;
       *imagp = 0.0;
       return(1);
     }

   /* second order tensor - change in z can't be > 2 */
   if ((abs(basis_set[ii].zcomponent - basis_set[jj].zcomponent)) > 2)
     {
       *realp = 0.0;
       *imagp = 0.0;
       return(1);
     }

   /* OK, we have a non-zero value */
   k = basis_set[ii].zcomponent - basis_set[jj].zcomponent;
   x = threeJ_a_ap1_2(II, i, -k);
   switch (II)
     {  /* multiply by reduced matrix element */
       case 1 : x *= (-4.0*sqrt(6.0));         /* (2 ||Q|| 1) */
                break;
       case 2 : x *= (-18.0);                  /* (3 ||Q|| 2) */
                break;
       case 3 : x *= (-10.0*sqrt(6.0));        /* (4 ||Q|| 3) */
                break;
       case 4 : x *= (-2.0*sqrt(3*5*11));      /* (5 ||Q|| 4) */
                break;
     }
```

```
   if ((II-i)%2) x = -x;          /* set sign */
   y = x * trQplr->ve[2-k].re;    /* include contraction */
   if (k%2) y = -y;               /* minus sign for contraction */
   *realp = y;                    /* return the value */
   y = x * trQplr->ve[2-k].im;
   if (k%2) y = -y;
   *imagp = y;
   return(1);

}

/*********************************************************************/

int matrix_element_7ov2(int ii, int jj, double *realp, double *imagp, ZVEC *trQplr, basis_element
*basis_set)
/*
 *  calculate matrix element using the Wigner-Eckart theorem for element ii, jj
 *  assume ii<jj, so superspins increase with basis set, as do z components
 *  the matrix elements ar of the form (Ii | 2k | Jj)
 *  according to Wigner = (-1)^(I-i) ( I 2 J) (I || 2 || J)
 *                                   (-i k j)
 *  we convert 3j symbol to (I J  2) with 2 minus signs, since J = I+1
 *                          (i j -k)
 *  otherwise matrix element is zero
 *  also include tensor contraction with rotated quadrupolar tensor, trQplr
 *  since this is the matrix element of operator with z component +k, we
 *  need the (-k) element of trQplr
 */

{
   int II, i, J, k;
   double x, y;

   II = basis_set[ii].superspin;
   i = basis_set[ii].zcomponent;
   J = basis_set[jj].superspin;

   /* quadrupole changes superspin by exactly 1 */
   if ((J - II) != 1)
     {
       *realp = 0.0;
       *imagp = 0.0;
       return(1);
     }

   /* second order tensor - change in z can't be > 2 */
   if ((abs(basis_set[ii].zcomponent - basis_set[jj].zcomponent)) > 2)
     {
       *realp = 0.0;
       *imagp = 0.0;
       return(1);
     }

   /* OK, we have a non-zero value */
   k = basis_set[ii].zcomponent - basis_set[jj].zcomponent;
   x = threeJ_a_ap1_2(II, i, -k);
   switch (II)
     {  /* multiply by reduced matrix element */
       case 1 : x *= (-6.0*sqrt(5.0));       /* (2 ||Q|| 1) */
                break;
       case 2 : x *= (-2.0*sqrt(3*5*11));    /* (3 ||Q|| 2) */
                break;
       case 3 : x *= (-12.0*sqrt(2*5));      /* (4 ||Q|| 3) */
                break;
       case 4 : x *= (-6.0*sqrt(5*13));      /* (5 ||Q|| 4) */
                break;
       case 5 : x *= (-14.0*sqrt(3*5));      /* (6 ||Q|| 5) */
                break;
       case 6 : x *= (-6.0*sqrt(2*5*7));     /* (7 ||Q|| 6) */
                break;
     }
```

```
   if ((II-i)%2) x = -x;          /* set sign */
   y = x * trQplr->ve[2-k].re;    /* include contraction */
   if (k%2) y = -y;               /* minus sign for contraction */
   *realp = y;                    /* return the value */
   y = x * trQplr->ve[2-k].im;
   if (k%2) y = -y;
   *imagp = y;
   return(1);

}

/*****************************************************************/

void norm_1(ZMAT *U)
/* normalize each eigenvector by its largest element */
{
    int i, j, jbig;
    double x, big;

    static ZVEC *temp = ZVNULL;

    temp = zv_resize(temp, U->m);

    for (i=0; i<U->n; i++)  /* for each column */
      {
        big = 0.0;   jbig = -1;
        for (j=0; j<U->m; j++)
          {
            temp->ve[j].re = U->me[j][i].re;
            temp->ve[j].im = U->me[j][i].im;
            x = zabs(U->me[j][i]);
            if (x > big)
              {
                jbig = j;
                big = x;
              }
          }  /* scanning the column */
        if (jbig < 0) printf("Error in normalization\n");
        /* now normalize */
        for (j=0; j<U->m; j++)
            U->me[j][i]= zdiv(temp->ve[j], temp->ve[jbig]);
      }
}

/*****************************************************************/

void mult_d2(double cbt, double sbt, ZVEC *in, ZVEC *out)
/* apply the second-order Wigner matrix for rotation by angle beta around y */
{
   static ZMAT *d2 = ZMNULL;

        d2 = zm_resize(d2, 5, 5);     /* second order Wigner matrix */
        zm_zero(d2);

           /* define Wigner elements */
        d2->me[2+2][2+2].re = 0.25 * SQR(1+cbt);
        d2->me[2+2][2+1].re = -0.5 *(1+cbt) * sbt;
        d2->me[2+2][2+0].re = sqrt(3.0/8.0) * SQR(sbt);
        d2->me[2+2][2-1].re = -0.5 *(1-cbt) * sbt;
        d2->me[2+2][2-2].re = 0.25 * SQR(1-cbt);

        d2->me[2+1][2+2].re = -d2->me[2+2][2+1].re;
        d2->me[2+1][2+1].re = 0.5 * (cbt-1) + SQR(cbt);
        d2->me[2+1][2+0].re = -sqrt(3.0/2.0) * sbt * cbt;
        d2->me[2+1][2-1].re = 0.5 * (cbt+1) - SQR(cbt);
        d2->me[2+1][2-2].re = d2->me[2+2][2-1].re;

        d2->me[2+0][2+2].re = d2->me[2+2][2+0].re;
        d2->me[2+0][2+1].re = -d2->me[2+1][2+0].re;
        d2->me[2+0][2+0].re = 0.5 * (3 * SQR(cbt) - 1);
        d2->me[2+0][2-1].re = d2->me[2+1][2+0].re;
```

```
        d2->me[2+0][2-2].re = d2->me[2+2][2+0].re;

        d2->me[2-1][2+2].re = -d2->me[2+2][2-1].re;
        d2->me[2-1][2+1].re = d2->me[2+1][2-1].re;
        d2->me[2-1][2+0].re = -d2->me[2+1][2+0].re;
        d2->me[2-1][2-1].re = d2->me[2+1][2+1].re;
        d2->me[2-1][2-2].re = d2->me[2+2][2+1].re;

        d2->me[2-2][2+2].re = d2->me[2+2][2-2].re;
        d2->me[2-2][2+1].re = -d2->me[2+2][2-1].re;
        d2->me[2-2][2+0].re = d2->me[2+2][2+0].re;
        d2->me[2-2][2-1].re = -d2->me[2+2][2+1].re;
        d2->me[2-2][2-2].re = d2->me[2+2][2+2].re;

        zmv_mlt(d2, in, out);
}

/*****************************************************************/

void mult_d1(double cbt, double sbt, ZVEC *in, ZVEC *out)
/* apply the first-order Wigner matrix for rotation by angle beta around y */
{
   static ZMAT *d1 = ZMNULL;

        d1 = zm_resize(d1, 3, 3);     /* first order Wigner matrix */
        zm_zero(d1);

           /* define Wigner elements */

        d1->me[1+1][1+1].re = 0.5 * (1+cbt);
        d1->me[1+1][1+0].re = -sbt/sqrt(2.0);
        d1->me[1+1][1-1].re = 0.5 * (1-cbt);

        d1->me[1+0][1+1].re = -d1->me[1+1][1+0].re;
        d1->me[1+0][1+0].re = cbt;
        d1->me[1+0][1-1].re = d1->me[1+1][1+0].re;

        d1->me[1-1][1+1].re = d1->me[1+1][1-1].re;
        d1->me[1-1][1+0].re = -d1->me[1+1][1+0].re;
        d1->me[1-1][1-1].re = d1->me[1+1][1+1].re;

        zmv_mlt(d1, in, out);
}

/*****************************************************************/

void mult_rz(double cal, double sal, ZVEC *in, ZVEC *out)
/* apply the second-order Wigner matrix for rotation by angle alpha around z */
{
   int n, n_central;
   static ZMAT *rz = ZMNULL;

    n = in->dim; /* dimension */
    n_central = (n-1)/2;

    rz = zm_resize(rz, n, n);     /* rotation around z, for second order */
    zm_zero(rz);

    if (n == 5)
      {
        rz->me[n_central+2][n_central+2].re = 2 * SQR(cal) - 1;   /* cos(2 alpha) */
        rz->me[n_central+2][n_central+2].im = 2 * sal * cal;      /* sin(2 alpha) */

        rz->me[n_central+1][n_central+1].re = cal;
        rz->me[n_central+1][n_central+1].im = sal;

        rz->me[n_central+0][n_central+0].re = 1.0;

        rz->me[n_central-1][n_central-1].re = cal;
        rz->me[n_central-1][n_central-1].im = -sal;
```

```
        rz->me[n_central-2][n_central-2].re = 2 * SQR(cal) - 1;
        rz->me[n_central-2][n_central-2].im = -2 * sal * cal;
      }

    if (n == 3)
      {
        rz->me[n_central+1][n_central+1].re = cal;
        rz->me[n_central+1][n_central+1].im = sal;

        rz->me[n_central+0][n_central+0].re = 1.0;

        rz->me[n_central-1][n_central-1].re = cal;
        rz->me[n_central-1][n_central-1].im = -sal;
      }

        zmv_mlt(rz, in, out);
}

/*********************************************************************/

void define_qbasis(basis_element *basis_set, int spinX2)
/*
 *  superspin starts at 1 (ignore 0) and increases
 *  within superspin, z component starts negative and ends up positive
 */
{
    int i, j, k;

    k=0;
    for (i=1; i<(spinX2+1); i++)
      {
        for (j=0; j<(2*i + 1); j++)
          {
            basis_set[k].superspin = i;
            basis_set[k].zcomponent = j-i;
            k++;
          }
      }
}

/*********************************************************************/

double threeJ_a_ap1_2(int a, int az, int cz)
{
    /*
     *  Wigner 3J symbol ( a   a+1   2  )
     *                   ( az -az-cz cz )
     *  use formulae in Brink and Satchler
     */

   double x, sign;

   sign = 1.0;
   if (cz < 0)  /* since sum on top is always odd, this will invert sign */
     {
       cz = -cz;
       az = -az;
       sign = -sign;
     }

   if (cz == 2)
     {
       x = sqrt((double) ((a+az+1)*(a+az+2)*(a+az+3)*(a-az))/((double)
(a*(a+1)*(2*a+4)*(2*a+3)*(2*a+1))));
       if ((a-az)%2) x = -x;
     }

   if (cz ==1)
     {
       x = sqrt((double) ((a+az+2)*(a+az+1))/((double) (a*(a+1)*(2*a+4)*(2*a+3)*(2*a+1))));
       x *= (a-2*az);
```

```
      if ((a-az-1)%2) x = -x;
    }

  if (cz == 0)
    {
      x = sqrt((double) (3*(a+az+1)*(a-az+1))/((double) (a*(a+1)*(a+2)*(2*a+3)*(2*a+1))));
      x *= az;
      if ((a-az-1)%2) x = -x;
    }

  x *= sign;

  return(x);
}

/*************************************************************************/


/*************************************************************************
*   This is the collection of routines that make up the complex general   *
*   eigenvalue routine                                                    *
*   actual declaration is way at the end                                  *
*                                                                         *
* void cg_(int nm, int n, double **ar, double **ai, double *wr,double *wi,*
*        int matz, double **zr, double **zi, double *fv1, double *fv2,    *
*        double *fv3, int *ierr)                                          *
*                                                                         *
*     on input                                                            *
*                                                                         *
*        nm  must be set to the row dimension of the two-dimensional      *
*        array parameters as declared in the calling program              *
*        dimension statement.                                             *
*                                                                         *
*        n  is the order of the matrix  a=(ar,ai).                        *
*                                                                         *
*        ar  and  ai  contain the real and imaginary parts,               *
*        respectively, of the complex general matrix.                     *
*                                                                         *
*        matz  is an int variable set equal to zero if                    *
*        only eigenvalues are desired.  otherwise it is set to            *
*        any non-zero int for both eigenvalues and eigenvectors.          *
*                                                                         *
*     on output                                                           *
*                                                                         *
*        wr  and  wi  contain the real and imaginary parts,               *
*        respectively, of the eigenvalues.                                *
*                                                                         *
*        zr  and  zi  contain the real and imaginary parts,               *
*        respectively, of the eigenvectors if matz is not zero.           *
*        The eigenvectors are stored as columns in zr and zi, so          *
*        that A * z = lamda * z                                           *
*                                                                         *
*        ierr  is an int output variable set equal to an error            *
*           completion code described in the documentation for comqr      *
*           and comqr2.  the normal completion code is zero.              *
*                                                                         *
*        fv1, fv2, and  fv3  are temporary storage arrays, of size n.     *
*                                                                         *
*************************************************************************/
/*************************************************************************
*                                                                        *
*         Copyright (c) 1993 SoftPulse Software. All rights reserved.    *
*                                                                        *
*************************************************************************/

/*
  Translated from the original Fortran into C by Tim Allman November 1, 1993.

  The source was translated into C using f2c.  The C code was then modified
  as follows:
```

```
  The call to cg() now uses the normal C pass-by-value convention.
  The array indexing was changed from column major to row major.
  The arrays use the pointers to pointers convention.  That is,
     the elements of the array are indexed through a vector of pointers
     to the rows of the arrays.  The array need not be contiguous
     when allocated.
  The complex arithmetic routines cadd, etc. only use pointers for the
     answer.  The input is passed by value.

  Beware!  Internally the 1 based array indexing was not changed.
           The passed pointers are decremented in cg() only.  If
           you wish to use one of the internal routines on its own
           you must decrement the pointers yourself.
*/
```