```
> # acm.mpl (version 1.4, 27 Sept 2015)
> #  - minor improvements over version 1.3 (details at end of file).

> ##################################################################
  ########

> # This code implements the ACM version of the Bohr model of the
> # atomic nucleus. The manuscript
> #   "A computer code for calculations in the algebraic collective
> #    model of the atomic nucleus",
> # by T.A. Welsh and D.J. Rowe [WR2015],
> # describes the mathematical foundations of the code, and also
> # serves as a manual. The manuscript (version 1.2) is available
  from
> #   http://arxiv.org/abs/1408.3824 (v2).
> # (A slightly improved version 1.3 was submitted for publication;
> #  this is being updated to version 1.4 following referees'
  comments.)

> # In brief, the code makes use of the SU(1,1) x SO(5) dynamical
> # group of the model, which enables the factorisation of states
> # into a direct product of radial states (with parameters anorm
> # and lambda) labelled by (nu), and SO(5) spherical harmonics,
> # labelled by (v,alpha,L,M).
> # We ignore the M throughout by dealing with reduced matrix
  elements.
> # The equation numbers, section numbers and tables referred to in
> # this file are those of the manuscript (version 1.4).
> # Note that the code makes use of Clebsch-Gordan coefficients
> # that are supplied in three zip archives. The files contained
> # in these archives should be unzipped and placed in a specific
> # directory stucture, as detailed in the manuscript.

> # Most of the calculations for the following paper [RWC2009] were
> # carried out using an earlier version of this code:
> #   "Bohr model as an algebraic collective model"
> #   by D.J. Rowe, T.A. Welsh and M.A. Caprio,
> #   Phys. Rev. C79 (2009) 054304.


> # The model was formulated in previous publications:

> # A pretty full explanation in Chapter 4 of the book [RowanWood]
> #   "Fundamentals of Nuclear Models: Foundational Models"
> #   by D.J. Rowe and J.L. Wood,
> #   World Scientific (Singapore), 2010.

> # An important precursor to this is the paper [Rowe2004]
> #   "A computationally tractable version of the collective model"
> #   by D.J. Rowe, Nucl. Phys. A 735 (2004) 372-392.

> ##################################################################
  ########

> # The code below is separated into eight parts:
> #   1. Specification of global constants, and procedures that
> #      can be used to set their values;
> #   2. Procedures that pertain only to the radial (beta) space;
> #   3. Procedures that access the SO(5)>SO(3) Clebsch-Gordon
  coefficients;
> #   4. Procedures that pertain only to the spherical (gamma,Omega)
  space;
```

```
> #   5. Procedures that obtain the internal representation of
  operators;
> #   6. Procedures that represent operators on the full (cross-
  product)
> #      Hilbert space;
> #   7. Procedures that perform calculations on the full Hilbert
  space:
> #      diagonalisating, basis transforming, and data displaying.
> #   8. Procedures that aid the production of the data for the
> #      particular Hamiltonians considered in [RWC2009].

> # Note that in the few occasions that I've used Maple's 'simplify'
> # function (usually to simplify expressions containing surds or
  GAMMA),
> # I've explicitly used the 'sqrt' or 'GAMMA' argument.
> # This has been necessary for some combinations of Maple version
  with
> # linux kernel (e.g. incompatibility between Maple15 and linux
  kernel 3.4.6).

> ##########################################################################
  ########
> ##########################################################################
  ########

> # Extensive use is made of the LinearAlgebra library.
> # In particular, this provides the diagonalisation procedure that
  we use.

> with(LinearAlgebra):


> ##########################################################################
  ########
> ####---------------------- Global Constants ----------------------
  ----####
> ##########################################################################
  ########

> # Here are specified various constants.
> # They should not be altered by the user.
> # They are mainly used to signify certain operators.
> # Hamiltonians and other operators will be expressed in terms of
> # these values.

> ACM_version:=1.4:

> # The following is a list containg the symbolic names for ten
  operators
> # that are the "basic" radial operators.
> # The way that they alter lambda is not fixed, but is determined
> # automatically.

> Radial_Operators:=[Radial_Sm, Radial_S0, Radial_Sp,
>                    Radial_b2, Radial_bm2, Radial_D2b, Radial_bDb,
>                    Radial_b, Radial_bm, Radial_Db]:

> # They will eventually be exchanged for operators in which the
  shift
> # is specific. The first seven keep their names (for zero shift),
> # but each instance of the final three will be exchanged for a
> # symbolic name that indicates a shift by a shift of -1,0 or +1.
```

```
> # The following lists will be used to achieve that.

> Radial_pl:=[Radial_b=Radial_b_pl,Radial_bm=Radial_bm_pl,
>             Radial_Db=Radial_Db_pl]:
> Radial_ml:=[Radial_b=Radial_b_ml,Radial_bm=Radial_bm_ml,
>             Radial_Db=Radial_Db_ml]:
> Radial_zl:=[Radial_b=Radial_b_zl,Radial_bm=Radial_bm_zl,
>             Radial_Db=Radial_Db_zl]:


> # The following indicates the SO(5) spherical harmonics for which
> # SO(5)>SO(3) Clebsch-Gordon coefficients are available,
> # and enables the v,alpha,L quantum numbers to be readily
> # obtained from the symbolic names.

> SpHarm_Table:=table([
>   SpHarm_010=[0,1,0],
>   SpHarm_112=[1,1,2],
>   SpHarm_212=[2,1,2], SpHarm_214=[2,1,4],
>   SpHarm_310=[3,1,0], SpHarm_313=[3,1,3], SpHarm_314=[3,1,4],
>   SpHarm_316=[3,1,6],
>   SpHarm_412=[4,1,2], SpHarm_414=[4,1,4], SpHarm_415=[4,1,5],
>   SpHarm_416=[4,1,6], SpHarm_418=[4,1,8],
>   SpHarm_512=[5,1,2], SpHarm_514=[5,1,4], SpHarm_515=[5,1,5],
>   SpHarm_516=[5,1,6], SpHarm_517=[5,1,7], SpHarm_518=[5,1,8],
>   SpHarm_51A=[5,1,10],
>   SpHarm_610=[6,1,0], SpHarm_613=[6,1,3], SpHarm_614=[6,1,4],
>   SpHarm_616=[6,1,6], SpHarm_626=[6,2,6], SpHarm_617=[6,1,7],
>   SpHarm_618=[6,1,8], SpHarm_619=[6,1,9], SpHarm_61A=[6,1,10],
>   SpHarm_61C=[6,1,12]
> ]):

> # Form a list of the available symbolic names in this table.

> SpHarm_Operators:=map(op,[indices(SpHarm_Table)]):

> # We also make use of SpDiag_sqLdim and SpDiag_sqLdiv which
> # denote operators represented by diagonal matrices with entries
> #     (-1)^{L_i}*sqrt(2L_i+1)
> # and (-1)^{L_i}/sqrt(2L_i+1) respectively.

> Spherical_Operators:=[op(SpHarm_Operators),SpDiag_sqLdim,
  SpDiag_sqLdiv]:

> # The four operators
> #       pi, [pi x pi]_{v=2,L=2}, [pi x pi]_{v=2,L=L}, [q x pi x pi]
  _{v=3,L=0}
> # intrinsically affect the whole product space:

> Xspace_Operators:=[ Xspace_Pi, Xspace_PiPi2, Xspace_PiPi4,
  Xspace_PiqPi ]:


> # The following quad_op specifies, in internal format, the
  quadrupole
> # operator. quadRigid_op is more appropriate for rigid-beta models.

> quad_op:=[ [Convert_112, [Radial_b,SpHarm_112]] ]:
> quadRigid_op:=[ [Convert_112, [SpHarm_112]] ]:
```

```
> # The following provide useful conversion factors from the SO(5)
> # spherical harmonics to more physically relevant operators;
> # see Table IV.
> # (Note that often (e.g. by RepSO5_Y_rem), the operator will be
  represented
> # with the 4*Pi already incorporated - and the FourPi should be
  cancelled).
> # Note that evalf will need to be used somewhere further down the
  line
> # to convert from these symbolic values to actual floating point
  values.

> FourPi:=4*Pi;
> Convert_112:=FourPi/sqrt(15);        # multiplies Y112 to get Q
> Convert_212:=-FourPi*sqrt(2/105);    # multiplies Y212 to get [QxQ]_
  (L=4)
> Convert_310:=FourPi/3;               # multiplies Y310 to get cos(3*
  gamma)
> Convert_316:=FourPi/3*sqrt(2/35);    # multiplies Y316 to get
  [QxQxQ]_(L=6)
> Convert_610:=2*FourPi/sqrt(15);      # multiplies Y610 to get [3*cos
  (3*gamma)+1]
> Convert_red:=1/FourPi;          # converts ME_SO5red to
  <v3|||v2|||v1>

> ##################################################################
  ########


> # The following procedure definitions determine functions used for
  displaying
> # the transition rates and amplitudes (the particular procedures in
  force at
> # a given time are stored in the global variables glb_rat_fun &
  glb_amp_fun).
> # These are as listed in Table V.
> # In addition to the matrix element Mel, they make use of the
  angular
> # momenta of the intitial state Li, and final state Lf.
> # (Maple doesn't allow me to specify a delimiting fourth argument $
  here!)

> # In the first of these procedures, the value of glb_rat_TRopAM
  (default 2),
> # is used to access the appropriate SO(3) CG coefficient:
> # glb_rat_TRopAM is the known angular momentum of the transition
  operator.

> quad_amp_fun:=proc(Li,Lf,Mel)
>   global glb_rat_TRopAM;
>   Mel*CG_SO3(Li,Li,glb_rat_TRopAM,Lf-Li,Lf,Lf)
> end;

> mel_amp_fun:=proc(Li,Lf,Mel)
>   Mel*sqrt(2*Lf+1)
> end;

> unit_amp_fun:=proc(Li,Lf,Mel)
>   Mel
> end;

> quad_rat_fun:=proc(Li,Lf,Mel)
```

```
>   Mel^2*dimSO3(Lf)/dimSO3(Li)
> end;

> mel_rat_fun:=proc(Li,Lf,Mel)
>   Mel^2*dimSO3(Lf)
> end;

> unit_rat_fun:=proc(Li,Lf,Mel)
>   Mel^2
> end;

> # The following was described in a previous version

> mix_amp_fun:=proc(Li,Lf,Mel)
>   global glb_rat_TRopAM;
>   Mel*gen_amp_mul(Li,Lf,glb_rat_TRopAM)
> end;

> gen_amp_mul:=proc(Li,Lf,Lt,$)
>   if Li=Lf then CG_SO3(Lf,Lf,Lt,0,Lf,Lf)
>   else sqrt(2*Lf+1)
>   fi:
> end;

> # The following procedure definitions determine functions used for
> # determining lambda as a function of seniority v
> # (the particular procedure in force at a given time is stored
> #  in the global variable glb_lam_fun: this is only accessed
> #  in the functions RepXspace_Twin, RepXspace_Pi, RepXspace_PiPi,
> #  RepXspace_PiqPi).
> # The first three are as listed in Table VI.
> # These functions return lambda_v-lambda_0 which must be an
  integer.
> # Analytic matrix elements result if the returned integer is of the
> # same parity as v (see Section 5.1).

> lambda_fix_fun:=proc(v::nonnegint)   # for fixed lambda
>   0
> end;

> lambda_sho_fun:=proc(v::nonnegint)   # for SHO lambda variation
>   v
> end;

> lambda_acm_fun:=proc(v::nonnegint)   # for lambda varying with
  parity of v
>   irem(v,2)
> end;

> lambda_jig_fun:=proc(v::nonnegint)   # A little mixture, used for
  testing
>   if v=0 then
>     0
>   else
>     2-irem(v,2)
>   fi
> end;

> # Further procedures of a similar nature may be obtained using the
> # following procedure, which returns the name of a procedure that
> # itself returns the nearest integer to
> #    sqrt( (v+3/2)^2 + C ) - sqrt( 9/4 + C )
```

```
> # of the same partity as v.
> # This is described in Appendix B.3.

> lambda_davi_fun:=proc(C::constant)
>   local difffun:

>     difffun:=proc(v::nonnegint) option operator,arrow;
>        local diffint:
>        diffint:=floor(sqrt((v+1.5)^2+C)-sqrt(2.25+C)):
>        if type(diffint-v,odd) then
>          diffint+1:
>        else
>          diffint:
>        fi:
>     end:

>    difffun:
> end:


> # We supply our own version of the square root that has `procedure`
  type.
> # It is necessary to use such a procedure to pass as an argument
> # when the type is being tested, because sqrt itself is not a
  `procedure`!
> # glb_amp_sft_fun:=sqrt:

> sqrt_fun:=proc(sft)
>   sqrt(evalf(sft)):
> end;


> ##################################################################
  ########

> # The SO(5)>SO(3) CG coefficients are initially obtained from
  external files.
> # The value of the Maple variable SO5CG_directory determines the
  directory
> # below which are to be found files containing SO(5)>SO(3) CG
  coefficients.
> # It may be specified at the start of a worksheet.
> # Or, if a acm-user.mpl file is used, it may be specified there.
> # A sample definition is (the final "/" is necessary):
> #     SO5CG_directory:="/home/username/maple/acm/so5cg-data/":  #
  sample

> # The procedure call
> #     show_CG_file(2,3,1,0,5):   # test
> # would test the directory specified in SO5CG_directory
> # (it is used by the procedure show_CG_file), and, somewhat, the
  data
> # therein (it should return two values: 0.522,0.431).

> # The following defines a table wherein the SO(5)>SO(3) Clebsch-
  Gordon
> # coefficients will be stored in memory. This table is intially
  empty.
> # The table is loaded from external files, as required.
> # For a particular (v1,v2,a2,L2,v3), this is done by calling
> # load_CG_table(v1,v2,a2,L2,v3).
> # When present, the SO(5)>SO(3) CG coefficient is given by
```

```
> # CG_coeffs[v1,v2,a2,L2,v3][a1,L1,a3,L3].

> CG_coeffs:=table():

> # To examine which (v1,v2,a2,L2,v3) have been loaded, we can use:
> #   indices(CG_coeffs);
> # Intially, of course, this table will be empty.

> ##################################################################
  ########

> # The following determine values used to set the defaults for how
  the
> # transition rates and amplitudes of the quadrupole operator are
> # displayed by the procedures Show_Rats and Show_Amps.

> def_rat_desg:="transition rates":
> def_rat_format:="  B(E2: %s) = %s":
> def_amp_desg:="transition amplitudes":
> def_amp_format:="  Amp( %s ) = %s":

> # If the Show_Mels procedure is used directly (Show_Rats and
> # Show_Amps call Show_Mels), the following two values can be used
> # (in fact, they are used by default).
> # These can also be used for the rates and amplitudes of other
  operators,
> # if the user hasn't defined anything else.

> def_mel_desg:="matrix elements":
> def_mel_format:="  ME( %s ) = %s":

> ##################################################################
  ########

> # The data that is produced by the main procedures is displayed
> # according to the values of various global parameters.
> # These are listed here, along with some initial values.
> # The values here should not be set directly, but by using the
> # ACM_set_ routines below.
> # Below, we use the ACM_set_defaults procedure which calls all
> # of the ACM_set_ procedures to set default values, overriding
> # the values given here.
> # (it may also be convenient to call ACM_set_ procedures from
> #  a file (acm-user.mpl) and read that into a Maple session.)

> # The following store the current factors used to divide
> # eigenvalues, transition rates and amplitudes displayed
> # respectively by the procedures Show_Eigs(), Show_Rats() and
> # Show_Amps(); and via these, by the procedures ACM_Scale()
> # and ACM_Adapt():

> glb_eig_sft:=1.0:
> glb_rat_sft:=1.0:
> glb_amp_sft:=1.0:

> # The following store the precision for floating point values that
> # are displayed by the procedures Show_Eigs(), Show_Rats() and
> # Show_Amps(); and via these, by the procedures ACM_Scale()
> # and ACM_Adapt():

> glb_rel_pre:=2:
> glb_rel_wid:=7:
```

```
> glb_low_pre:=4:

> # The following store the maximal number of entries for horizontal
> # lists of eigenvalues, transition rates and amplitudes that are
> # respectively displayed by the procedures Show_Eigs(), Show_Rats()
> # and Show_Amps(); and via these, by the procedures ACM_Scale()
> # and ACM_Adapt():

> glb_eig_num:=4:
> glb_rat_num:=4:
> glb_amp_num:=4:

> # The following specify how ACM_Adapt() determines the scale factor
> # glb_eig_sft. This factor is determined such that the energy of
> # the (glb_eig_idx)th state of AM glb_eig_L comes out to be
  glb_eig_fit.

> glb_eig_fit:=6.0:
> glb_eig_L:=2:
> glb_eig_idx:=1:

> # The following specify how ACM_Adapt() determines the scale factor
> # glb_rat_sft. This factor is determined such that the transition
  rate
> # from the (glb_rat_1dx)th state of AM glb_eig_L1 to the
> # (glb_rat_2dx)th state of AM glb_eig_L2 comes out to be
  glb_rat_fit.

> glb_rat_fit:=100.0:
> glb_rat_L1:=2:
> glb_rat_L2:=0:
> glb_rat_1dx:=1:
> glb_rat_2dx:=1:

> # The following specifies a procedure which determines the basis
  type.
> # This is a function which gives the value of lambda_v-lambda_0.

> glb_lam_fun:=lambda_acm_fun:

> # The following store the current transition operator and its
> # angular momentum.
> # The former is the operator for which transition rates and
> # amplitudes are calculated in the procedures ACM_Scale() and
  ACM_Adapt().
> # The latter is used in two (minor) ways:
> #   1. by Show_Mels() (via Show_Rats & Show_Amps) so that only
  those
> #      lists for which |Lf-Li| <= glb_rat_TRopAM are
> #      output, for otherwise the MEs are zero;
> #   2. in a couple of the predefined procedures above
> #      (mix_amp_fun & quad_amp_fun).

> glb_rat_TRop:=quad_op:
> glb_rat_TRopAM:=2:

> # The following determine how "transition rates" are displayed in
  the
> # procedure Show_Rats (which is called by ACM_Scale and ACM_Adapt).
> # The first specifies the formula used, the second the format used
  to
> # display each value, the third the phrase used to designate the
```

```
  values.
> # (e.g. "transition rates")

> glb_rat_fun:=quad_rat_fun:
> glb_rat_format:=def_rat_format:
> glb_rat_desg:=def_rat_desg:

> # The following determine how "transition rates" are displayed in
  the
> # procedure Show_Amps (which is called by ACM_Scale and ACM_Adapt).
> # The first specifies the formula used, the second the format used
  to
> # display each value, the third the phrase used to designate the
  values.
> # (e.g. "transition amplitudes")

> glb_amp_fun:=quad_amp_fun:
> glb_amp_format:=def_amp_format:
> glb_amp_desg:=def_amp_desg:

> # The following specifies the function by which the scaling factor
> # for transition amplitudes (glb_amp_sft) is obtained from that
> # (glb_rat_sft) for transition rates:

> glb_amp_sft_fun:=sqrt:

> # The following determines how the matrix element labels are
  displayed:

> glb_tran_format:="%s(%s) -> %s(%s)":
> glb_tran_fill:="#":

> # The following store the lists of transition rate and transition
  amplitude
> # designators (each initially empty):

> glb_rat_lst:=[]:
> glb_amp_lst:=[]:

> # The following flag indicates whether, in ACM_Scale, ACM_Adapt
> # and Show_Eigs, eigenvalues are displayed relative to their
> # lowest value (true), or absolute (false).

> glb_eig_rel:=true:


> # The following parameter, if positive, specifies a temporary
> # increase to the size of the radial space, to improve accuracy
> # of radial reps.
> # It is used only by the procedure RepXspace_Twin (which is called
> # by RepXspace).

> glb_nu_lap:=0:

> ##################################################################
  ########

> # We now give a set of procedures that specify values of the above
> # parameters. The last of these, ACM_set_defaults, uses the others
> # to set values for all of the above parameters.
> # In each of these procedures, if the final passed argument is 1
> # then the procedure prints out a brief description of its effect.
```

```
> # The return value contains the now current values of all the
> # parameters that the procedure can set.

> # The following sets the values of glb_eig_sft, glb_rat_sft,
  glb_amp_sft
> # Note that the scaling factor glb_amp_sft is obtained from
  glb_rat_sft
> # using the procedure given by glb_amp_sft_fun.

> ACM_set_scales:=proc(eig_sft::constant,rat_sft::constant,
>                      show::integer:=1,$)
>       global glb_eig_sft,glb_rat_sft,glb_amp_sft,glb_amp_sft_fun;

>   if _npassed>0 then
>     glb_eig_sft:=evalf(eig_sft);
>   fi:
>   if _npassed>1 then
>     glb_rat_sft:=evalf(rat_sft);
>   fi:

>   glb_amp_sft:=glb_amp_sft_fun(glb_rat_sft);  #default is square
  root

>   ACM_show_scales(show):
> end;

> # The following displays and returns the values of the scaling
  factors
> # glb_eig_sft, glb_rat_sft, glb_amp_sft.

> ACM_show_scales:=proc(show::integer:=1,$)
>       global glb_eig_sft,glb_rat_sft,glb_amp_sft,
>              glb_rat_desg,glb_amp_desg:

>   if show>0 then
>     printf("Relative eigenenergies to be multiplied by %f;\n",
>                 evalf(1/glb_eig_sft));
>     printf("\"%s\" to be multiplied by %f;\n",
>                 glb_rat_desg,evalf(1/glb_rat_sft));
>     printf("\"%s\" to be multiplied by %f.\n",
>                 glb_amp_desg,evalf(1/glb_amp_sft));
>   fi:

>   [glb_eig_sft,glb_rat_sft,glb_amp_sft]:
> end;

> # The following sets glb_amp_sft_fun, and returns NULL:

> ACM_set_sft_fun:=proc(amp_fun::procedure:=glb_amp_sft_fun,
>                       show::integer:=1,$)
>     global glb_amp_sft_fun,glb_amp_desg;

>   glb_amp_sft_fun:=amp_fun;

>   if show>0 then
>       printf("\"%s\" scaling factor calculated"
>                 " using the procedure: \"%a\".\n",
>                   glb_amp_desg,
>                   glb_amp_sft_fun):
>   fi:
>
>   glb_amp_sft_fun:
```

```
> end;

> # The following sets the values of glb_rel_pre, glb_rel_wid, and
  glb_low_pre

> ACM_set_output:=proc(rel_pre::nonnegint,rel_wid::nonnegint,
  low_pre::nonnegint,
>                      show::integer:=1,$)
>     global glb_low_pre,glb_rel_wid,glb_rel_pre;

>   if _npassed>0 then
>     glb_rel_pre:=rel_pre;
>   fi:
>   if _npassed>1 then
>     glb_rel_wid:=rel_wid;
>   fi:
>   if _npassed>2 then
>     glb_low_pre:=low_pre;
>   fi:

>   if show>0 then
>   printf("%d decimal places for each displayed value,\n",
  glb_rel_pre);
>   printf("%d total digits for each displayed value,\n",glb_rel_wid)
  ;
>   printf("except %d decimal places for lowest (absolute)
  eigenvalue.\n",
>                                                        glb_low_pre)
  ;
>   fi:

>   [glb_rel_pre, glb_rel_wid, glb_low_pre]:
> end;

> # The following sets the values of glb_eig_num, glb_rat_num, and
> # glb_amp_num. For simplicity, the latter two are set equal.

> ACM_set_listln:=proc(eig_num::nonnegint,rat_num::nonnegint,
>                      show::integer:=1,$)
>       global glb_eig_num,glb_rat_num,glb_amp_num;

>   if _npassed>0 then
>     glb_eig_num:=eig_num;
>   fi:
>   if _npassed>1 then
>     glb_rat_num:=rat_num;
>     glb_amp_num:=rat_num;
>   fi:

>   if show>0 then
>   printf("Display lowest %d eigenvalue(s) at each L.\n",
  glb_eig_num);
>   printf("Display lowest %d rate/amplitude(s) in each list.\n",
  glb_rat_num);
>   fi:

>   [glb_eig_num, glb_rat_num, glb_amp_num]:
> end;

> # The following sets the boolean value of glb_eig_rel.

> ACM_set_datum:=proc(datflag::nonnegint:=1,
```

```
>                        show::integer:=1,$)
>       global glb_eig_rel:

>   glb_eig_rel:=evalb(datflag>0):

>   if show>0 then
>     if glb_eig_rel then
>       printf("Eigenvalues displayed relative to minimal value.\n")
>     else
>       printf("Absolute eigenvalues displayed.\n")
>     fi:
>   fi:

>   glb_eig_rel:
> end;

> # The following sets the values of glb_eig_fit, glb_eig_L,
  glb_eig_idx,
> # which are used by ACM_Adapt to determine the factor glb_eig_sft.

> ACM_set_eig_fit:=proc(eig_fit::constant:=glb_eig_fit,
>                       eig_L::nonnegint:=glb_eig_L,
>                       eig_idx::posint:=1, show::integer:=1,$)
>       global glb_eig_fit, glb_eig_L, glb_eig_idx;

>   glb_eig_fit:=evalf(eig_fit);
>   glb_eig_L:=eig_L;
>   glb_eig_idx:=eig_idx;

>   if show>0 then
>     printf("In ACM_Adapt, the scaling factor for relative
  eigenvalues "
>            "is chosen such that\nthat for the %d(%d) state is
  %f\n",
>                                        glb_eig_L,glb_eig_idx,
  glb_eig_fit);
>   fi:

>   [glb_eig_fit, glb_eig_L, glb_eig_idx]:
> end;

> # Similarly, the following sets the values of
> #      glb_rat_fit, glb_rat_L1, glb_rat_1dx, glb_rat_L2,
  glb_rat_2dx:
> # which are used by ACM_Adapt to scale the transition rates output.

> ACM_set_rat_fit:=proc(rat_fit::constant:=glb_rat_fit,
>                       rat_L1::nonnegint:=glb_rat_L1,
>                       rat_L2::nonnegint:=glb_rat_L2,
>                       rat_1dx::posint:=1,
>                       rat_2dx::posint:=1,
>                       show::integer:=1,$)
>       local tran_fmat,rat_fmat,rat_this:
>       global glb_rat_fit, glb_rat_L1, glb_rat_1dx, glb_rat_L2,
  glb_rat_2dx,
>              glb_rat_format, glb_rat_desg, glb_tran_format:

>     glb_rat_fit:=evalf(rat_fit);
>     glb_rat_L1:=rat_L1;
>     glb_rat_L2:=rat_L2;

>     glb_rat_1dx:=rat_1dx;
```

```
>     glb_rat_2dx:=rat_2dx;

>   if show>0 then

>   # Change the %s specifications in glb_tran_fmat to "%d" for
  integers.

>     tran_fmat:=sprintf(glb_tran_format,"%d","%d","%d","%d"):
>     rat_fmat:=sprintf(glb_rat_format,tran_fmat,"%f"):
>     rat_this:=sprintf(rat_fmat,glb_rat_L1,glb_rat_1dx,
>                               glb_rat_L2,glb_rat_2dx,glb_rat_fit):

>     printf("In ACM_Adapt, the scaling factor for \"%s\" "
>                                "is chosen such that\n%s\n",
>            glb_rat_desg,rat_this);
>   fi:

>   [glb_rat_fit, glb_rat_L1, glb_rat_L2, glb_rat_1dx, glb_rat_2dx]:
> end;


> # The following three functions respectively set, augment or
  display the
> # list rat_lst, which determines which transition rates are flagged
> # for display. If no argument is given for the first two, an empty
  list
> # is assumed.

> ACM_set_rat_lst:=proc(rat_lst::list(list(integer)):=[],$)
>     global glb_rat_lst;

>   glb_rat_lst:=[];
>   ACM_add_rat_lst(rat_lst):
> end;

> #

> ACM_add_rat_lst:=proc(rat_lst::list(list(integer)):=[],$)
>     local rat_ent;
>     global glb_rat_lst;

>   for rat_ent in rat_lst do
>     if nops(rat_ent)>5 then
>       printf("  Bad transition rate specification: %a\n",rat_ent):
>     else
>       glb_rat_lst:=[op(glb_rat_lst),rat_ent]:
>     fi:
>   od:

>   return nops(glb_rat_lst);
> end;

> #

> ACM_show_rat_lst:=proc(show::integer:=1,$)
>       local rate_ent,rat_format4,rat_format5;
>       global glb_rat_lst,glb_tran_format,glb_rat_desg;

>   if show>0 then

>     if nops(glb_rat_lst)>0 then
>       rat_format4:=sprintf(glb_tran_format,"%d","%d","%d","%d"):
```

```
>       printf("Following \"%s\" are set to be displayed:\n",
  glb_rat_desg):
>       for rate_ent in glb_rat_lst do

>         if nops(rate_ent)=4 or (nops(rate_ent)=5 and rate_ent[5]=0)
  then
>             printf("  "):
>             printf(rat_format4,
>                rate_ent[1], rate_ent[3], rate_ent[2], rate_ent[4]):
>             printf("\n"):
>         elif nops(rate_ent)=5 then
>             printf("  "):
>             rat_format5:=sprintf(glb_tran_format,
>                                  "%d%+dk","%d","%d%+dk","%d"):
>             printf(rat_format5, rate_ent[1], rate_ent[5], rate_ent
  [3],
>                                  rate_ent[2], rate_ent[5], rate_ent
  [4]):
>             printf("\n"):
>         elif nops(rate_ent)=3 then
>             rat_format5:=sprintf(glb_tran_format, "%d","j_i","%d",
  "%d"):
>             printf("  "):
>             printf(rat_format5, rate_ent[1], rate_ent[2], rate_ent
  [3]):
>             printf("\n"):
>         elif nops(rate_ent)=2 then
>             printf("  "):
>             rat_format5:=sprintf(glb_tran_format, "%d","j_i","%d",
  "j_f"):
>             printf(rat_format5, rate_ent[1], rate_ent[2]):
>             printf("\n"):
>         elif nops(rate_ent)=1 then
>             printf("  "):
>             rat_format5:=sprintf(glb_tran_format, "L_i","j_i","%d",
  "j_f"):
>             printf(rat_format5, rate_ent[1]):
>         elif nops(rate_ent)=0 then
>             printf("  "):
>             printf(glb_tran_format, "L_i","j_i","L_f","j_f"):
>             printf("\n"):
>         fi

>       od
>     else
>       printf("Currently, no \"%s\" are set to be displayed.\n",
  glb_rat_desg):
>     fi:
>   fi:

>   return glb_rat_lst;
> end;

> # The following three functions respectively set, augment or
  display the
> # list amp_lst, which determines which transition amplitudes are
  flagged
> # for display. If no argument is given for the first two, an empty
  list
> # is assumed.

> ACM_set_amp_lst:=proc(amp_lst::list(list(integer)):=[],$)
```

```
>     global glb_amp_lst;

>   glb_amp_lst:=[];
>   ACM_add_amp_lst(amp_lst):
> end;

> #

> ACM_add_amp_lst:=proc(amp_lst::list(list(integer)):=[],$)
>     local amp_ent;
>     global glb_amp_lst;

>   for amp_ent in amp_lst do
>     if nops(amp_ent)>5 then
>       printf("  Bad amplitude specification: %a\n",amp_ent):
>     else
>       glb_amp_lst:=[op(glb_amp_lst),amp_ent]:
>     fi:
>   od:

>   return nops(glb_amp_lst);
> end;

> #

> ACM_show_amp_lst:=proc(show::integer:=1,$)
>       local amp_ent,amp_format4,amp_format5;
>       global glb_amp_lst,glb_tran_format,glb_amp_desg;

>   if show>0 then

>     if nops(glb_amp_lst)>0 then
>       amp_format4:=sprintf(glb_tran_format,"%d","%d","%d","%d"):

>       printf("Following \"%s\" are set to be displayed:\n",
  glb_amp_desg):
>       for amp_ent in glb_amp_lst do

>         if nops(amp_ent)=4 or (nops(amp_ent)=5 and amp_ent[5]=0)
  then
>             printf("   "):
>             printf(amp_format4,
>                 amp_ent[1], amp_ent[3], amp_ent[2], amp_ent[4]):
>             printf("\n"):
>         elif nops(amp_ent)=5 then
>             printf("   "):
>             amp_format5:=sprintf(glb_tran_format,
>                           "%d%+dk","%d","%d%+dk","%d"):
>             printf(amp_format5, amp_ent[1], amp_ent[5], amp_ent[3],
>                                 amp_ent[2], amp_ent[5], amp_ent[4])
  :
>             printf("\n"):
>         elif nops(amp_ent)=3 then
>             printf("   "):
>             amp_format5:=sprintf(glb_tran_format,
>                           "%d","j_i","%d","%d"):
>             printf(amp_format5, amp_ent[1], amp_ent[2], amp_ent[3])
  :
>             printf("\n"):
>         elif nops(amp_ent)=2 then
>             printf("   "):
>             amp_format5:=sprintf(glb_tran_format,
```

```
>                               "%d","j_i","%d","j_f"):
>             printf(amp_format5, amp_ent[1], amp_ent[2]):
>             printf("\n"):
>         elif nops(amp_ent)=1 then
>             printf("  "):
>             amp_format5:=sprintf(glb_tran_format,
>                               "L_i","j_i","%d","j_f"):
>             printf(amp_format5, amp_ent[1]):
>         elif nops(amp_ent)=0 then
>             printf("  "):
>             printf(glb_tran_format, "L_i","j_i","L_f","j_f"):
>             printf("\n"):
>         fi

>       od
>     else
>       printf("Currently, no \"%s\" are set to be displayed.\n",
  glb_amp_desg):
>     fi:
>   fi:

>   return glb_amp_lst;
> end;


> # The following specifies the transition rate operator
  glb_rat_TRop.
> # It also attempts to determine its angular momentum
  glb_rat_TRopAM.

> ACM_set_transition:=proc(TR_op::list(list):=glb_rat_TRop,
>                          show::integer:=1,$)
>     local rat_AM:
>     global glb_rat_TRop,glb_rat_TRopAM,glb_rat_desg;

>   glb_rat_TRop:=TR_op;
>   rat_AM:=Op_AM(TR_op):
>   glb_rat_TRopAM:=abs(rat_AM):   # this is largest value of AM, if
  LC.

>   if show>0 then
> #      printf("In ACM_Scale and ACM_Adapt, \"%s\" "
>       printf("In ACM_Scale and ACM_Adapt, transition matrix
  elements "
>                  "now calculated for the operator:\n",glb_rat_desg)
  :
>       print( glb_rat_TRop):

>       if rat_AM>=0 then
>         printf("(This has angular momentum %a).\n\n",rat_AM):
>       else
>         printf("(This has indeterminate angular momentum: "
>                                    "maximum %a).\n\n",-rat_AM):
>       fi:
>   fi:

>   [glb_rat_TRop,glb_rat_TRopAM]:
> end;


> # The following sets glb_rat_fun, glb_rat_format, and glb_rat_desg
> # which determine how "transition rates" are displayed in the
```

```
> # procedure Show_Rats (which is called by ACM_Scale and ACM_Adapt).
> # These values are displayed if the final fourth argument is 1
  (default).

> ACM_set_rat_form:=proc(rat_fun::procedure:=glb_rat_fun,
>                        rat_format::string:=glb_rat_format,
>                        rat_desg::string:=glb_rat_desg,
>                        show::integer:=1,$)

>       global glb_rat_fun,glb_rat_format,glb_rat_desg,
  glb_tran_format;
>       local tran_fmat1;

>   glb_rat_fun:=rat_fun:
>   glb_rat_format:=rat_format:
>   glb_rat_desg:=rat_desg:

>   if show>0 then
>     printf("ACM_Scale and ACM_Adapt now set to display \"%s\"
  first.\n",
>                 glb_rat_desg):
>     printf("These are calculated from the (alternative reduced)
  transition"
>              " matrix elements\nusing the procedure: \"%a\".\n",
>                 glb_rat_fun):

>     tran_fmat1:=sprintf(glb_tran_format,"L_i","j_i","L_f","j_f");
>     printf("Each will be output using the format:\n  ");
>     printf(glb_rat_format,tran_fmat1,"*"):
>     printf("\n");
>   fi:

>   [glb_rat_fun,glb_rat_format,glb_rat_desg]:
> end;


> # The following sets glb_amp_fun, glb_amp_format, and glb_amp_desg
> # which determine how "transition amplitudes" are displayed in the
> # procedure Show_Amps (which is called by ACM_Scale and ACM_Adapt).
> # These values are displayed if the final fourth argument is 1
  (default).

> ACM_set_amp_form:=proc(amp_fun::procedure:=glb_amp_fun,
>                        amp_format::string:=glb_amp_format,
>                        amp_desg::string:=glb_amp_desg,
>                        show::integer:=1,$)

>       global glb_amp_fun,glb_amp_format,glb_amp_desg,
  glb_tran_format;
>       local tran_fmat1;

>   glb_amp_fun:=amp_fun:
>   glb_amp_format:=amp_format:
>   glb_amp_desg:=amp_desg:

>   if show>0 then
>     printf("ACM_Scale and ACM_Adapt now set to display \"%s\"
  second.\n",
>                 glb_amp_desg):
>     printf("These are calculated from the (alternative reduced)
  transition"
>              " matrix elements\nusing the procedure: \"%a\".\n",
```

```
>                 glb_amp_fun):

>     tran_fmat1:=sprintf(glb_tran_format,"L_i","j_i","L_f","j_f");
>     printf("Each will be output using the format:\n  ");
>     printf(glb_amp_format,tran_fmat1,"*"):
>     printf("\n");
>   fi:

>   [glb_amp_fun,glb_amp_format,glb_amp_desg]:
> end;


> # The following specfies the "basis type" procedure glb_lam_fun.
> # (see also next procedure).

> ACM_set_lambda_fun:=proc(lambda_fun::procedure, show::integer:=1,$)
>     global glb_lam_fun;
>   glb_lam_fun:=lambda_fun:

>   if show>0 then
>       printf("lambda values calculated from v using the "
>                   "procedure: \"%a\",\n", glb_lam_fun):
>   fi:

>   glb_lam_fun:
> end;

> # The following uses the above procedure to set glb_lam_fun to one
  of
> # four particular basis types, using procedures defined elsewhere.
> # These basis types are those specified in (63), (61), (62), (B17)
  resp.
> # For choice=0, lambda_v=lambda_0,
> #     choice=1, lambda_v=lambda_0 + v,
> #     choice=2, lambda_v=lambda_0 + (v mod 2),
> #     choice=3, lambda_v=lambda_0 + "integer Davidson variation",
> # the latter obtained using lambda_davi_fun().
> # The second argument is used only if the first is 3.

> ACM_set_basis_type:=proc(choice::nonnegint, abeta0::constant:=0.0,
>                                                 show::integer:=1,
  $)
>   local new_fun:
>   global glb_lam_fun, lambda_fix_fun, lambda_sho_fun,
>                          lambda_acm_fun, lambda_jig_fun:
>   if choice=0 then
>     if show>0 then
>       printf("Using the constant lambda basis.\n"):
>     fi:
>     ACM_set_lambda_fun(lambda_fix_fun,0):
>   elif choice=1 then
>     if show>0 then
>       printf("Using the harmonic oscillator basis with "
>                   "lambda_v = lambda_0 + v.\n"):
>     fi:
>     ACM_set_lambda_fun(lambda_sho_fun,0):
>   elif choice=2 then
>     if show>0 then
>       printf("Using the ACM parity basis.\n"):
>     fi:
>     ACM_set_lambda_fun(lambda_acm_fun,0):
>   elif choice=3 then
```

```
>     if show>0 then
>       printf("Using integer Davidson basis for potential with "
>              "minimum at %a (dimensionless).\n",abeta0):
>     fi:
>     new_fun:=lambda_davi_fun(abeta0^4):
>     ACM_set_lambda_fun(new_fun,0):
>   else
>     error "There is no basis %1 defined!", choice:
>   fi:

> end;


> # For the currently defined basis stored in glb_lam_fun, the
  following
> # returns lambda_v-lambda_0 for v=vmin...vmax.

> ACM_show_lambda_fun:=proc(vmin::nonnegint:=0,vmax::nonnegint:=10)
>   global glb_lam_fun;
>   [seq(glb_lam_fun(v),v=vmin..vmax)]:
> end;


> # Following tests that lambda only shifts by +/-1 as we change v,
> # returning boolean true if so, false if not.
> # (This procedure is not used elsewhere.)

> # ACM_test_lambda_fun:=proc(vmin::nonnegint,vmax::nonnegint)
> #   local v,lam,lamv:
> #   global glb_lam_fun:
> #   lam:=glb_lam_fun(vmin):
> #   for v from vmin+1 to vmax do
> #     lamv:=glb_lam_fun(v):
> #     if lamv-lam=1 or lamv-lam=-1 then
> #       lam:=lamv:
> #     else
> #       return false:
> #     fi:
> #   od:
> #
> #   true:
> # end;


> # The following procedure calls the above procedures to set the
> # default values for all of the global parameters described above.
> # Note that the location of the SO(5)>SO(3) Clebsch-Gordon
  coefficients
> # must also be specified somewhere by setting the variable
  SO5CG_directory.

> ACM_set_defaults:=proc(show::integer:=1)

>   ACM_set_output(2,7,4,show):
>   ACM_set_listln(4,4,show):
>   ACM_set_datum(1,show):
>   ACM_set_basis_type(2,0.0,show):

>   ACM_set_transition(quad_op,show):
>   ACM_set_rat_form(quad_rat_fun,def_rat_format,def_rat_desg,show):
>   ACM_set_amp_form(quad_amp_fun,def_amp_format,def_amp_desg,show):
>   ACM_set_sft_fun(sqrt_fun,show):
```

```
>   ACM_set_eig_fit(6.0,2,1,show):
>   ACM_set_rat_fit(100.0,2,0,1,1,show):
>   ACM_set_rat_lst( [] ):
>   ACM_set_amp_lst( [] ):
>   ACM_show_rat_lst(show):
>   ACM_show_amp_lst(show):
>   ACM_set_scales(1.0,1.0,show):

>   if show>0 then
>     printf("\n"):
>   fi:

> end:
>


> ########################################################################
  ########
> ####--------------- Representations on the radial space -----------
  ----####
> ########################################################################
  ########

> # The next set of routines deal with representing operators in the
> # radial (beta) space. The bases for the radial Hilbert space are
> # dependent on two parameters (a,lambda). For each such pair,
> # the basis states are labelled by a single index nu=0,1,2,....

> # A truncated Hilbert space is indexed by states labelled
> #     nu_min, nu_min+1, nu_min+2, ... nu_max
> # (usually we would use nu_min=0).
> # The following two functions each take arguments nu_min and
  nu_max;
> # the first returns the dimension of the truncated space,
> # the second returns a list of all the labels.

> dimRadial:=(nu_min::nonnegint,nu_max::nonnegint)
>   -> `if`(nu_max>=nu_min,nu_max-nu_min+1,0):


> lbsRadial:=proc(nu_min::nonnegint,nu_max::nonnegint)
>   if nu_min>nu_max then
>     error("Radial range invalid");
>   else
>     [seq(i,i=nu_min..nu_max)];
>   fi:
> end:


> ########################################################################
  ########

> # The functions that follow calculate single matrix elements
> #     F^{(a)}_{lambda',mu_f}{lambda,mu_i}(Op),
> # as defined by (13), for various operators Op between two
> # radial space states labelled by non-negative integers mu_i and
  mu_f.
> # Note that lambda and lambda' might not be equal, and thus the
  states
> # belong to different bases (in each of the following procedures,
> # lambda'-lambda is a certain fixed value (mostly 0,+1 or -1);
> # also note that we require both lambda>1 and lambda'>1).
```

```
> # These routines return the matrix elements for a=1 (more general
> # values are obtained later by multiplying by a power of a).
> # The type of the return value is float only if that of lambda is.

> # The following three give matrix elements of the SU(1,1) operators
  S0,S+,S-.
> # These use eqns. (16)-(18).

> ME_Radial_S0:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i then
>     lambda/2 + mu_i;
>   else
>     0;
>   fi;
> end:

> ME_Radial_Sp:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i+1 then
>     sqrt( (lambda + mu_i)*(mu_i+1) );
>   else
>     0;
>   fi;
> end:

> ME_Radial_Sm:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i-1 then
>     sqrt( (lambda + mu_i - 1)*mu_i );
>   else
>     0;
>   fi;
> end:


> # The following give matrix elements of beta^2 for lambda'=lambda
> # using (21).

> ME_Radial_b2:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i-1 then
>     sqrt( (lambda + mu_i - 1)*mu_i );
>   elif mu_f=mu_i then
>     lambda + 2*mu_i;
>   elif mu_f=mu_i+1 then
>     sqrt( (lambda + mu_i)*(mu_i+1) );
>   else
>     0;
>   fi;
> end:

> # The following gives matrix elements of 1/beta^2 for lambda'=
  lambda
> # using (22). It uses the subsequent procedure for which mu_f >=
  mu_i.
> # (restriction to lambda>1).

> ME_Radial_bm2:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if lambda=-1 then
>     error "Singular 1/beta^2 for lambda=1";
```

```
>   fi:
>   if frac(lambda)=0 and (lambda <= -mu_i or lambda <= -mu_f) then
>     error "cannot evaluate Gamma function at non-positive integer":
>   fi:

>   if mu_f>=mu_i then
>     ME_Radial_pt(lambda,mu_f,mu_i);
>   else
>     ME_Radial_pt(lambda,mu_i,mu_f);
>   fi:
> end:

> ME_Radial_pt:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   (-1)^(mu_f-mu_i) * sqrt( (factorial(mu_f)*GAMMA(lambda+mu_i))
>                                /(factorial(mu_i)*GAMMA(lambda+mu_f)
  ) )
>                  / (lambda-1);
> end:

> # The following gives matrix elements of d^2/d(beta)^2 for lambda'=
  lambda
> # using (23).

> ME_Radial_D2b:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   local stuff:
>   if mu_f=mu_i-1 then
>     stuff:=sqrt( (lambda + mu_i - 1)*mu_i );
>   elif mu_f=mu_i then
>     stuff:=-lambda - 2*mu_i;
>   elif mu_f=mu_i+1 then
>     stuff:=sqrt( (lambda + mu_i)*(mu_i+1) );
>   else
>     stuff:=0;
>   fi;

>   if mu_f>=mu_i then
>     stuff+(lambda-(3/2))*(lambda-(1/2))*ME_Radial_pt(lambda,mu_f,
  mu_i);
>   else
>     stuff+(lambda-(3/2))*(lambda-(1/2))*ME_Radial_pt(lambda,mu_i,
  mu_f);
>   fi:
> end:

> # The following gives matrix elements of beta*d/d(beta) for
  lambda'=lambda
> # using (24).

> ME_Radial_bDb:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i-1 then
>     sqrt( (lambda + mu_i - 1)*mu_i );
>   elif mu_f=mu_i then
>     -(1/2);
>   elif mu_f=mu_i+1 then
>     -sqrt( (lambda + mu_i)*(mu_i+1) );
>   else
>     0;
>   fi;
> end:
```

```
> # The following gives matrix elements of beta for lambda'=lambda+1
> # using (26).

> ME_Radial_b_pl:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if mu_f=mu_i-1 then
>     sqrt( mu_i );
>   elif mu_f=mu_i then
>     sqrt(lambda + mu_i);
>   else
>     0;
>   fi;
> end:

> # The following gives matrix elements of 1/beta for lambda'=
  lambda+1
> # using (28).

> ME_Radial_bm_pl:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if frac(lambda)=0 and lambda <= -mu_i then
>     error "cannot evaluate Gamma function at non-positive integer":
>   fi:

>   if mu_f<mu_i then
>     0;
>   else
>     (-1)^(mu_f-mu_i)*sqrt( (factorial(mu_f)*GAMMA(lambda+mu_i))
>                           /(factorial(mu_i)*GAMMA(lambda+mu_f+1))
  );
>   fi:
> end:

> # The following gives matrix elements of d/d(beta) for lambda'=
  lambda+1
> # using (30).

> ME_Radial_Db_pl:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   local res:
>   if mu_f=mu_i-1 then
>     res:=sqrt( mu_i );
>   elif mu_f=mu_i then
>     res:=-sqrt( lambda + mu_i );
>   else
>     res:=0;
>   fi;

>   if mu_f>=mu_i then
>     res:=res+(-1)^(mu_f-mu_i) * (lambda-1/2)
>                * sqrt( (factorial(mu_f)*GAMMA(lambda+mu_i))
>                          /(factorial(mu_i) * GAMMA(lambda+mu_f+1))
  );
>   fi:
>   res:
> end:

> # The following gives matrix elements of beta for lambda'=lambda-1
> # using (27).

> ME_Radial_b_ml:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
```

```
>   if mu_f=mu_i+1 then
>     sqrt( mu_f );
>   elif mu_f=mu_i then
>     sqrt(lambda + mu_i - 1);
>   else
>     0;
>   fi;
> end:

> # The following gives matrix elements of 1/beta for lambda'=
  lambda-1
> # using (29).

> ME_Radial_bm_ml:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint)
>   if frac(lambda)=0 and lambda <= -mu_i then
>     error "cannot evaluate Gamma function at non-positive integer":
>   fi:

>   if mu_f>mu_i then
>     0;
>   else
>     (-1)^(mu_f-mu_i)*sqrt( (factorial(mu_i)*GAMMA(lambda+mu_f-1))
>                           /(factorial(mu_f)*GAMMA(lambda+mu_i)) );
>   fi:
> end:

> # The following gives matrix elements of d/d(beta) for lambda'=
  lambda-1
> # using (31).

> ME_Radial_Db_ml:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint) local res:
>   if mu_f=mu_i+1 then
>     res:=-sqrt( mu_f );
>   elif mu_f=mu_i then
>     res:=sqrt(lambda + mu_i - 1);
>   else
>     res:=0;
>   fi;

>   if mu_f<=mu_i then
>     res:=res+(-1)^(mu_f-mu_i) * (3/2-lambda)
>                * sqrt( (factorial(mu_i)*GAMMA(lambda+mu_f-1))
>                          /(factorial(mu_f)*GAMMA(lambda+mu_i)) );
>   fi:
>   res:
> end:

> # The following gives matrix elements of the identity operator
> # for lambda'=lambda+2r, for nonnegative r, using (33).
> # It makes use of MF_Radial_id_poly below.

> ME_Radial_id_pl:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint,
>                                     r::nonnegint)
>   if frac(lambda)=0 and lambda <= -mu_i then
>     error "cannot evaluate Gamma function at non-positive integer":
>   fi:

>   if mu_i<=mu_f+r then
>     eval(MF_Radial_id_poly(mu_f,mu_i,r),lamvar=lambda)
```

```
>             *sqrt( (factorial(mu_f)*GAMMA(lambda+mu_i))
>                       /(factorial(mu_i)*GAMMA(lambda+mu_f+2*r)) )
>   else
>     0
>   fi:
> end:

> # The following gives matrix elements of the identity operator
> # for lambda'=lambda-2r, for nonnegative r, using (33).
> # It makes use of MF_Radial_id_poly below.

> ME_Radial_id_ml:=proc(lambda::algebraic,mu_f::nonnegint,
  mu_i::nonnegint,
>                                     r::nonnegint)
>   if frac(lambda)=0 and lambda <= -mu_f+2*r then
>     error "cannot evaluate Gamma function at non-positive integer":
>   fi:

>   if mu_f<=mu_i+r then
>     eval(MF_Radial_id_poly(mu_i,mu_f,r),lamvar=lambda-2*r)
>             *sqrt( (factorial(mu_i)*GAMMA(lambda+mu_f-2*r))
>                       /(factorial(mu_f)*GAMMA(lambda+mu_i)) )
>   else
>     0
>   fi:
> end:

> # The following, used by the above two procedures, calculates (33)
> # for all non-negative integer r. It returns a polynomial in
  lamvar.
> # Note that this works for r=0 (giving delta_{mu,nu}, as required).

> MF_Radial_id_poly:=proc(mu::nonnegint,nu::nonnegint,r::nonnegint)
>   local res:

>   if nu>mu+r then
>     return(0):
>   fi:

>   res:=add( (-1)^j * binomial(r,j) * binomial(r+mu-nu+j-1,r-1)
>               * GAMMA(lamvar+mu+2*r)/GAMMA(lamvar+mu+r+j)
>               * GAMMA(mu+j+1)/GAMMA(mu+1),
>                                                 j=max(0,nu-mu)..r):

>   simplify(res,GAMMA)*(-1)^(mu+nu):
> end;

> # Old version of above, which evaluates at the particular value
> # of lambda.

> MF_Radial_id_pl:=proc(lambda::algebraic,mu::nonnegint,
  nu::nonnegint,
>                                   r::nonnegint)
>   local res:

>   if nu>mu+r then
>     return(0):
>   fi:

>   res:=add( (-1)^j * binomial(r,j) * binomial(r+mu-nu+j-1,r-1)
>               * GAMMA(lambda+mu+2*r)/GAMMA(lambda+mu+r+j)
>               * GAMMA(mu+j+1)/GAMMA(mu+1),
```

```
>                                                  j=max(0,nu-mu)..r):

>   simplify(res,GAMMA)*(-1)^(mu+nu):
> end;

> # Same result, but done in a different way.

> MF_Radial_id_pl2:=proc(lambda::algebraic,mu::nonnegint,
  nu::nonnegint,
>                                    r::nonnegint)
>   local res:

>   if nu>mu+r then
>     return(0):
>   fi:

>   res:=add( (-1)^j * binomial(r,j) * binomial(2*r+mu-nu-j-1,r+mu-
  nu)
>               * GAMMA(lambda+2*r+mu)/GAMMA(lambda+r-1)
>               * GAMMA(lambda+2*r-j-1)/GAMMA(lambda+2*r+mu-j),
>                                                  j=0..k-1):

>   if nu=mu+r then
>     res:=res+ (-1)^r * GAMMA(lambda+2*r+mu)/GAMMA(lambda+r+mu):
>   fi:

>   simplify(res,GAMMA)*(-1)^(mu+nu):
> end;


> # The following procedure returns a single matrix element
> #     F^{(anorm)}_{lambda_var,mu_f}{lambda,mu_i}(Op),
> # for Op one of the operators from Table I with symbolic name
  radial_op.
> # The identity operator is also available using symbolic name
  Radial_id.
> # If possible, the matrix element is obtained using one of the
> # above procedures. If not, it is obtained by matrix multiplication
> # where one matrix is obtained non-analytically.
> # This procedure is not used elsewhere.

> ME_Radial:=proc(radial_op::algebraic, anorm::algebraic,
>                   lambda::algebraic, lambda_var::integer,
>                   mu_f::nonnegint, mu_i::nonnegint)
>   local MM:

>   if radial_op=Radial_b2 and lambda_var=0 then
>     ME_Radial_b2(lambda,mu_f,mu_i)/anorm^2;
>   elif radial_op=Radial_bm2 and lambda_var=0 then
>     ME_Radial_bm2(lambda,mu_f,mu_i)*anorm^2;
>   elif radial_op=Radial_D2b and lambda_var=0 then
>     ME_Radial_D2b(lambda,mu_f,mu_i)*anorm^2;
>   elif radial_op=Radial_bDb and lambda_var=0 then
>     ME_Radial_bDb(lambda,mu_f,mu_i);

>   elif radial_op=Radial_b and lambda_var=1 then
>     ME_Radial_b_pl(lambda,mu_f,mu_i)/anorm;
>   elif radial_op=Radial_bm and lambda_var=1 then
>     ME_Radial_bm_pl(lambda,mu_f,mu_i)*anorm;
>   elif radial_op=Radial_Db and lambda_var=1 then
>     ME_Radial_Db_pl(lambda,mu_f,mu_i)*anorm;
```

```
>   elif radial_op=Radial_b and lambda_var=-1 then
>     ME_Radial_b_ml(lambda,mu_f,mu_i)/anorm;
>   elif radial_op=Radial_bm and lambda_var=-1 then
>     ME_Radial_bm_ml(lambda,mu_f,mu_i)*anorm;
>   elif radial_op=Radial_Db and lambda_var=-1 then
>     ME_Radial_Db_ml(lambda,mu_f,mu_i)*anorm;

>   elif radial_op=Radial_S0 and lambda_var=0 then
>     ME_Radial_S0(lambda,mu_f,mu_i);
>   elif radial_op=Radial_Sp and lambda_var=0 then
>     ME_Radial_Sp(lambda,mu_f,mu_i);
>   elif radial_op=Radial_Sm and lambda_var=0 then
>     ME_Radial_Sm(lambda,mu_f,mu_i);


>   elif radial_op=Radial_id and type(lambda_var,even) then
>     if lambda_var>=0 then
>       ME_Radial_id_pl(lambda,mu_f,mu_i,lambda_var/2):
>     else
>       ME_Radial_id_ml(lambda,mu_f,mu_i,-lambda_var/2):
>     fi:

>   else  # form a matrix
>         # might be a good idea to check that we have a valid radial
  operator
>       if radial_op=Radial_id then
>         MM:=RepRadial_Prod([],_passed[2..4],0,max(mu_f,mu_i),
>                                       iquo(abs(lambda_var)+3,2)):
>       else
>         MM:=RepRadial_Prod([radial_op],_passed[2..4],0,max(mu_f,
  mu_i),
>                                       iquo(abs(lambda_var)+3,2)):
>       fi:
>       MM[mu_f+1,mu_i+1]:  # matrices start at mu=nu=0!
>   fi:

> end:

> ##################################################################
  ########

> # The following uses one of the above procedures
> #    ME_Radial_S0, ME_Radial_Sp, ME_Radial_Sm,
> #    ME_Radial_b2, ME_Radial_bm2, ME_Radial_D2b, ME_Radial_bDb,
> #    ME_Radial_b_pl, ME_Radial_bm_pl, ME_Radial_Db_pl,
> #    ME_Radial_b_ml, ME_Radial_bm_ml, ME_Radial_Db_ml
> # above, this being specified in the first argument, to construct
  an
> # explicit representation matrix from the elements
> #     F^{(a)}_{lambda',mu_f}{lambda,mu_i}(Op),
> # nu_min <= mu_i,mu_f <= nu_max, where Op is the corresponding
  operator,
> # and lambda'-lambda is as above. This is for a=1: the general a
  case
> # is obtained later by multiplying by some power of a.

> # Note that the datatype of the resulting matrix is not fixed:
> # Maple chooses it depending on the type of lambda
> # (e.g. lambda=5/2 gives algebraic, lambda=2.5 gives floats;
> #  these may be tested for using the Maple commands
> #          type(MM,'Matrix'(datatype=anything));
> #          type(MM,'Matrix'(datatype=float));
```

```
> # ).
> # It'd thus be a good idea to apply evalf to all the matrix
  elements
> # obtained before diagonalisation etc.

> # Typically, this procedure and the two that follow will get called
> # many times during the construction of an operator (Hamiltonian)
  for
> # various values of lambda, but the same range of nu_min & nu_max.
> # Each use the remember option, and these remember tables are
  cleared
> # at the end of the procedures RepXspace and RepRadial_Prod.

> RepRadial:=proc(ME::procedure,lambda::algebraic,
>                                 nu_min::nonnegint,nu_max::nonnegint)
>     option remember;

>   simplify(Matrix(nu_max-nu_min+1,(i,j)->ME(lambda,nu_min-1+i,
  nu_min-1+j)),
>         GAMMA,radical):
> end:

> # The following works similarly to RepRadial above, but takes an
  additional
> # parameter which is passed to the procedure ME which calculates
  the
> # matrix elements. This enables the construction of representations
  of
> # the identity operator using the procedures ME_Radial_id_pl and
> # ME_Radial_id_ml.

> RepRadial_param:=proc(ME::procedure,lambda::algebraic,
>                             nu_min::nonnegint,nu_max::nonnegint,
  param::integer)
>     option remember;

>   simplify(Matrix(nu_max-nu_min+1,
>                   (i,j)->ME(lambda,nu_min-1+i,nu_min-1+j,param)),
>         GAMMA,radical):
> end:


> # The following returns the square root of the matrix obtained
  above.
> # The arguments are as above, and the return matrix contain float
  entries.
> # (This has severe problems dealing with Matrices larger than about
  20x20 -
> #  the problem is in Maple's MatrixPower).
> # This has now been replaced by Matrix_sqrt below

> #RepRadial_sq:=proc(ME::procedure,lambda::algebraic,
> #                                 nu_min::nonnegint,
  nu_max::nonnegint)
> #    option remember;
> #
> #  MatrixPower(evalf(RepRadial(ME,lambda,nu_min,nu_max)),1/2):
> #end:


> # The following returns the positive definite square root of a
> # symmetric Matrix, using my Eigenfiddle procedure (defined later)
```

```
> # which provides a convenient interface to Maple's Eigenvectors
  procedure.

> Matrix_sqrt:=proc(Amatrix::Matrix,$)
>     option remember;
>     local Edata,Diag_sq:

>     # first obtain (real) eigenvalues and eigenvectors

>     Edata:=Eigenfiddle(evalf(Amatrix)):

>     # then form diagonal Matrix from square roots of eigenvalues

>     Diag_sq:=Matrix(map(sqrt,Edata[1]),scan=diagonal);

>     # transform back into the original (non eigen) basis

>     Edata[2].Diag_sq.MatrixInverse(Edata[2])
> end:

> # The following is similar to the above to produce the inverse of
> # the square root of a Matrix.

> Matrix_sqrtInv:=proc(Amatrix::Matrix,$)
>     option remember;
>     local Edata,Diag_sq:

>     # first obtain (real) eigenvalues and eigenvectors

>     Edata:=Eigenfiddle(evalf(Amatrix)):

>     # then form diagonal Matrix from square roots of eigenvalues

>     Diag_sq:=Matrix(map(x->1/sqrt(x),Edata[1]),scan=diagonal);

>     # transform back into the original (non eigen) basis

>     Edata[2].Diag_sq.MatrixInverse(Edata[2])
> end:


> ##################################################################
  ########

> # The following represents the radial operator beta^K * d^T/d(beta)
  ^T,
> # with a specific lambda shift (for K integer, T nonneg integer, R
  integer).
> # It returns the explicit matrix of elements
> #     F^{(anorm)}_{lambda+R,mu_f}{lambda,mu_i}(beta^K * d^T/d(beta)
  ^T),
> # nu_min <= mu_i,mu_f <= nu_max. It is only used by
  RepRadialshfs_Prod().
> # The values lambda and lambda+R should be positive (an error
> # results if this is not the case).

> # This is implemented by forming a product between matrices for
> # the terms beta and d/d(beta) (some are paired, e.g. beta^2),
> # and splitting R amongst these terms in a certain judicious way,
> # with each getting a lambda shift of +1,0,-1,
> # and also using the identity operator with a shift if required
> # (this splitting is determined by the procedure Lambda_Splits
```

```
  below).

> # The matrix elements of the result are analytic (exact expressions
> # involving surds) unless anorm or lambda are floats, or K+T+R is
  odd,
> # in which cases the matrix elements might be a mix of floats and
  surds.

> RepRadial_bS_DS:=proc(K::integer, T::nonnegint, anorm::algebraic,
>                           lambda::algebraic, R::integer,
>                           nu_min::nonnegint, nu_max::nonnegint)
>   option remember;
>   local i,n,imm,Mat,Mat_product,lambda_run,lamX,lam_splits;

>   if evalf(lambda)<=0 or evalf(lambda+R)<=0 then
>      error("Non-positive lambda shift for operator [%1,%2]",K,T):
>   fi:

>   # deal first with the special case that K=T=0 and R is odd:

>   if K=0 and T=0 and type(R,odd) then
>     if R<0 then  # beta[0] * (1/beta)[-1] * id[-even]

>       Mat_product:=RepRadial(ME_Radial_b2,lambda+R,nu_min,nu_max):
>       Mat_product:=Matrix_sqrt(Mat_product):

>       Mat:=RepRadial(ME_Radial_bm_ml,lambda+R+1,nu_min,nu_max);
>       Mat_product:=MatrixMatrixMultiply(Mat_product,Mat):

>       if R<-1 then
>         Mat:=RepRadial_param(ME_Radial_id_ml,lambda,nu_min,nu_max,-
  (R+1)/2);
>         Mat_product:=MatrixMatrixMultiply(Mat_product,Mat):
>       fi:

>     else  # id[even] * (1/beta)[-1] * beta[0]

>       Mat_product:=RepRadial(ME_Radial_b2,lambda,nu_min,nu_max):
>       Mat_product:=Matrix_sqrt(Mat_product):

>       Mat:=RepRadial(ME_Radial_bm_pl,lambda,nu_min,nu_max);
>       Mat_product:=MatrixMatrixMultiply(Mat,Mat_product):

>       if R>1 then
>         Mat:=RepRadial_param(ME_Radial_id_pl,lambda+1,nu_min,
  nu_max,(R-1)/2);
>         Mat_product:=MatrixMatrixMultiply(Mat,Mat_product):
>       fi:

>     fi:

>     return (Mat_product):
>   fi:


>   # determine how to partition the lambda shift R amongst the
  individual terms
>   # (we could do this in-line)

>   lam_splits:=Lambda_Splits(K,T,R):

>   # note that we have to account for there possibly being excess
```

```
  variation,
>   # this being the case if there are more entries in lam_splits
  than |K|+T.
>   # In such a case, lam_splits[1] should be even becase the only
  possible
>   # odd case (see Lambda_Splits() above) arises for K=T=0 and R
  odd,
>   # and this has already been dealt with.

>   n:=abs(K)+T:

>   if nops(lam_splits)>n then
>     lamX:=lam_splits[1]:            # even extra variation - for
  identity op
>     lam_splits:=lam_splits[2..-1]:  # remove first element

>     # below we then prepend or append the identity operator;

>   else
>     lamX:=0:
>   fi:

>   # we now work right to left building up the product, with the
>   # current value of lambda being carried along
>   # (the procedure Lambda_Splits better deals with R->L).

>   lambda_run:=lambda:

>   # We put an identity op on the right if lamX<0 (on left for
  lamX>0 below)

>   if lamX<0 then
>     Mat_product:=RepRadial_param(ME_Radial_id_ml,lambda_run,nu_min,
  nu_max,
>                                                        -lamX/2);
>     lambda_run:=lambda_run+lamX:
>   fi:

>   # form required product, multiplying from the right, and changing
  the
>   # lambdas as we go. First set up loop.


>   i:=n:

>   while i>0 do
>
>     if lam_splits[i]>0 then

>       if i<=K then  # then K is +ve
>         Mat:=RepRadial(ME_Radial_b_pl,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,1/anorm);

>       elif i<=-K then  # then K is -ve
>         Mat:=RepRadial(ME_Radial_bm_pl,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm);

>       else # then i>|K|
>         Mat:=RepRadial(ME_Radial_Db_pl,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm);
>       fi:
```

```
>       imm:=1:

>     elif lam_splits[i]<0 then

>       if i<=K then  # then K is +ve => beta
>         Mat:=RepRadial(ME_Radial_b_ml,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,1/anorm);

>       elif i<=-K then  # then K is -ve => beta^{-1}
>         Mat:=RepRadial(ME_Radial_bm_ml,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm);

>       else # then i>|K|
>         Mat:=RepRadial(ME_Radial_Db_ml,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm);
>       fi:

>       imm:=1:

>     elif i>1 and lam_splits[i-1]=0 then   # pair 00 of lambda
  changers

>       if i<=K then # then K is +ve => beta^2
>         Mat:=RepRadial(ME_Radial_b2,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,1/anorm^2);

>       elif i<=-K then  # then K is -ve => beta^{-2}
>         Mat:=RepRadial(ME_Radial_bm2,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm^2);

>       elif i=K+1 then # then K is +ve => beta*d/d(beta)
>         Mat:=RepRadial(ME_Radial_bDb,lambda_run,nu_min,nu_max);


>       elif i=-K-1 then # then K is -ve => beta^{-1}*d/d(beta)
>         error("This shouldn't arise!"):

>       else # d^2/d(beta)^2
>         Mat:=RepRadial(ME_Radial_D2b,lambda_run,nu_min,nu_max);
>         Mat:=MatrixScalarMultiply(Mat,anorm^2);
>       fi:

>       imm:=2:

>     else # an isolated 0 lambda change

>       if i<=K then  # then K is +ve => beta
>         # obtain a matrix representing beta by taking the positive
>         # definite square root of that representing beta^2.
>         Mat:=RepRadial(ME_Radial_b2,lambda_run,nu_min,nu_max):
>         Mat:=Matrix_sqrt(Mat):
>         Mat:=MatrixScalarMultiply(Mat,1/anorm);

>       elif i<=-K then  # then K is -ve => beta^{-1}
>         # obtain a matrix representing 1/beta by taking the
>         # inverse of the positive definite square root of that
>         # representing beta^2.
>         Mat:=RepRadial(ME_Radial_b2,lambda_run,nu_min,nu_max):
>         Mat:=Matrix_sqrtInv(Mat):
>         Mat:=MatrixScalarMultiply(Mat,anorm);

>       else # then i>|K| => d/d(beta)
```

```
>         # obtain a matrix representing d/d(beta) by taking the
  inverse
>         # of the positive definite square root of that representing
  beta^2
>         # multiplied by that for beta*d/d(beta).
>         Mat:=RepRadial(ME_Radial_b2,lambda_run,nu_min,nu_max):
>         Mat:=MatrixMatrixMultiply( Matrix_sqrtInv(Mat),
>                 evalf(RepRadial(ME_Radial_bDb,lambda_run,nu_min,
  nu_max))):
>         Mat:=MatrixScalarMultiply(Mat,anorm);
>       fi:

>       imm:=1:

>     fi:

>     # multiply this term into product

>     if i=n and lamX>=0 then  # first in product
>       Mat_product:=Mat:
>         # These matrices now have the same storage: but Mat_product
  is not
>         # then changed when Mat is reassigned to another Matrix in
  the next
>         # instance of loop.
>     else
>         # It would be nice to use inplace multiplication here, but
  this
>         # fails when the two matrices have entries of different
  types.
>       Mat_product:=MatrixMatrixMultiply(Mat,Mat_product):
>     fi:

>     lambda_run:=lambda_run+lam_splits[i]:   # update lambda along
  product
>     i:=i-imm: # skip index to next op

>   od:


>   # We still might need to multiply on the left by an even lambda>0
  shifted
>   # identity operator.

>   if lamX>0 then
>     Mat:=RepRadial_param(ME_Radial_id_pl,lambda_run,nu_min,nu_max,
  lamX/2);
>     if n>0 then
>       Mat_product:=MatrixMatrixMultiply(Mat,Mat_product):
>     else
>       Mat_product:=Mat:
>     fi:
>   fi:

>   # In the following case, nothing has yet been formed:

>   if n=0 and lamX=0 then
>     Mat_product:=Matrix([seq(1,i=nu_min..nu_max)],scan=diagonal):
>   fi:

>   # Maple sometimes has problems unless we specify the type of
  combine...
```

```
>   combine(simplify(Mat_product, sqrt),radical):

> end:

> # The following procedure is (only) called by the above
  RepRadial_bS_DS:
> # it considers a term of the form beta^K * d^T/d(beta)^T, and for
> # a specific overall lambda shift R, indicates how to sensibly
  assign
> # lambda shifts of 0,+1 or -1 to each term
> # (there are various constraints - for one, the zero shifts should
> # be paired apart from one case which we put at start; another is
> # that a beta*d/d(beta) is split first (because the beta might be
  beta^(-1)).
> # The return is a list of integers of length |K|+T or |K|+T+1.
> # In the former case, it is just the list of shifts;
> # in the latter case, an extra entry is put at the start:
> # the calling procedure is required to test for this.
> # This value is the shift required for an extra identity operator
> # (it is even in as many cases as possible -
> #          but is odd in only one case: iff K=T=0 and R is odd).

> Lambda_Splits:=proc(K::integer, T::nonnegint, R::integer)
>   local KT,IR,Z,ZT,shifts:

>   KT:=abs(K)+T:  # shiftings to be assigned
>   IR:=abs(R)-KT: # +ve if we need extra lambda shift

>   if IR>0 then   # put extra shift at start
>     if type(IR,even) then
>       shifts:=[IR,1$KT]:
>     elif KT>0 then # cannot do this if KT=0
>       shifts:=[IR+1,0,1$(KT-1)]:
>     else
>       shifts:=[IR]: # odd flag - extra processsing needed later
>     fi:
>   else
>     Z:=iquo(-IR,2):  # no of zero pairs to be assigned
>     ZT:=min(Z,iquo(T,2)):  # no of these for the Ts
>     shifts:=[0$(KT-abs(R)-2*ZT),1$abs(R),0$(2*ZT)]:
>     # special case to prevent beta^(-1)*d/d(beta) being assigned
  00.
>     if K<0 and R=0 and type(T,odd) then
>       shifts[-K]:=1:
>       shifts[-K+1]:=-1:
>     fi:
>   fi:

>   if R<0 then
>     -shifts
>   else
>     shifts
>   fi:

> end:


> # The following returns, for a certain Op determined by rps_op,
> # the explicit matrix of elements
> #     F^{(anorm)}_{lambda+R,mu_f}{lambda,mu_i}(Op),
> # nu_min <= mu_i,mu_f <= nu_max.
```

```
> # Here Op is a product formed from beta, d/d(beta), and the su(1,1)
> # operators Sp,Sm,S0. In rps_op, it is given as a list of elements
> # of types [K,T] and S, the former obtained using RepRadial_bS_DS
  below
> # and the latter directly from RepRadial (here K,T and S are
  integers,
> # with T nonnegative, and S=+1,-1,0).
> # For each element in the list rps_op, the lambda shift is
  specified
> # by the corresponding element of lambda_shfs (the two lists should
> # then be the same size).

> RepRadialshfs_Prod:=proc(rps_op::list, anorm::algebraic,
>                          lambda::algebraic, lambda_shfs::list,
>                          nu_min::nonnegint, nu_max::nonnegint)

>   option remember;
>   local i,n,Mat,Mat_product,lambda_run,r_op;

>   n:=nops(rps_op);

>   if n=0 then    # null product: require identity matrix
>     Mat_product:=Matrix([seq(1,i=nu_min..nu_max)],scan=diagonal):

>   else
>     lambda_run:=lambda:

>     # form required product, multiplying from the right
>     # with inplace multiplications ...

>     for i from n to 1 by -1 do

>       r_op:=rps_op[i]:

>       if type(r_op,integer) then
>         if lambda_shfs[i]<>0 then
>           error(
>             "Non-zero lambda shift for S operator (this shouldn't
  arise!)"):
>         fi:

>         if r_op=0 then
>           Mat:=RepRadial(ME_Radial_S0,lambda_run,nu_min,nu_max);
>         elif r_op=1 then
>           Mat:=RepRadial(ME_Radial_Sp,lambda_run,nu_min,nu_max);
>         elif r_op=-1 then
>           Mat:=RepRadial(ME_Radial_Sm,lambda_run,nu_min,nu_max);
>         else
>           error("Unrecognised S operator"):
>         fi:
>       elif type(r_op,list(integer)) then
>         # r_op[1]:   integer exponent of beta
>         # r_op[2]:   non-neg integer, order of d/d(beta)

>         Mat:=RepRadial_bS_DS(r_op[1],r_op[2],anorm,lambda_run,
  lambda_shfs[i],
>                                    nu_min,nu_max):
>         lambda_run:=lambda_run+lambda_shfs[i]:
>       else
>           error "radial operator %1 undefined", r_op;
>       fi:
```

```
>       if i=n then
>         Mat_product:=Mat:
>           # These matrices now have the same storage: but
  Mat_product is not
>           # then changed when Mat is reassigned to another Matrix
  in the next
>           # instance of loop.
>       else
>         Mat_product:=MatrixMatrixMultiply(Mat,Mat_product):
>       fi:

>     od:

>   fi:

>   # Maple sometimes has problems unless we specify the type of
  combine...

>   combine(simplify(Mat_product, sqrt),radical):

> end;



> # The following represents a product Op of radial operators,
  specified by a
> # list rbs_op, between two bases with the difference between their
  lambda
> # values given by lambda_var. It returns the explicit matrix of
> # elements
> #     F^{(anorm)}_{lambda+lambda_var,mu_f}{lambda,mu_i}(Op),
> # for nu_min <= mu_i,mu_f <= nu_max.
> # rbs_op is a list of symbolic names of the "basic"
  Radial_Operators
> #     Radial_b2, Radial_bm2, Radial_D2b, Radial_bDb,
> #     Radial_b,  Radial_bm, Radial_Db,
> #     Radial_S0, Radial_Sp, Radial_Sm,
> # (for beta^2; 1/beta^2; d^2/d(beta)^2; beta*d/d(beta);
> #  beta; 1/beta; d/d(beta); S0; S+; S-; respectively).

> # The result might need evalf operating on it to ensure that the
  returned
> # matrix has float entries.
> # The matrix elements of the result are analytic (exact expressions
> # involving surds) unless anorm or lambda are floats, or the parity
> # of the operator rbs_op is opposite to that of lambda_var,
> # in which cases the matrix elements might be a mix of floats and
  surds.

> # This procedure might, via RepRadialshfs_Prod, eventually call any
> # of the procedures
> #      RepRadial, RepRadial_param, RepRadial_bS_DS,
> #      Matrix_sqrt and Matrix_sqrtInv,
> # each of which uses a remember option.
> # We provide two versions, the first of which clears those remember
  table:
> # this is the one described in the manual (it is called only by
  ME_Radial
> # in this code.)
> # Note that they use Parse_RadialOp_List and Lambda_RadialOp_List
  below.
```

```
> # Implementation details:
> # The list rbs_op is parsed to split it up (using the procedure
> # Parse_RadialOp_List) into sequences of operators of the form
> # (beta^K * d/d(beta)^T) and S+, S-, S0.
> # Then, the variation lambda_var in lambda is split (using the
> # procedure Lambda_RadialOp_List) amongst these operators
> # (an extra operator 1 might get prepended at this point).
> # The representations of the operators with these lambda
> # variations is then obtained using RepRadialshfs_Prod
> # (this gets reps for S+, S-, S0 directly, but calls
> # RepRadial_bS_DS for (beta^K * d/d(beta)^T) ).
> # If lambda_var is of the same parity as rbs_op, then
> # the result will be analytic (however truncation effects
> # during matrix multiplication might affect the accuracy of the
> # outlying matrix elements). Otherwise, somewhere along the
> # line, a matrix square root is taken and this results in
> # floating point matrix elements, or combinations of such and
  surds.


> RepRadial_Prod:=proc(rbs_op::list, anorm::algebraic,
>                          lambda::algebraic, lambda_var::integer,
>                          nu_min::nonnegint, nu_max::nonnegint,
>                          nu_lap::nonnegint:=0,$)
>     local parsed_ops,lambda_shfs,rep,nu_min_shift:

>     if evalf(lambda)<=0 then
>        error("Non-positive lambda value %1",lambda):
>     elif evalf(lambda+lambda_var)<=0 then
>        error("Non-positive lambda value %1",lambda+lambda_var):
>     fi:

>     # parse into a list of [K,T] and S ops (S=0,+1,-1)

>     parsed_ops:=Parse_RadialOp_List(rbs_op):

>     # assign lambda shifts to each of [K,T] and S.

>     lambda_shfs:=Lambda_RadialOp_List(parsed_ops,lambda_var):

>     # if resulting list is one larger than given, then need to
>     # prepend or append a [0,0] operator depending on whether
>     # the extra variation is positive or negative

>     if nops(lambda_shfs)>nops(parsed_ops) then
>       if lambda_shfs[1]>0 then
>         parsed_ops:=[ [0,0], op(parsed_ops) ]:
>       else
>         parsed_ops:=[ op(parsed_ops), [0,0] ]:
>         lambda_shfs:=[op(2..-1,lambda_shfs),lambda_shfs[1]]:
>       fi:
>     fi:
>
>     # the indices of the matrix must be extended by at most nu_lap
  in
>     # both directions

>     nu_min_shift:=min(nu_lap,nu_min):

>     # now find representation of this product, with given
  lambda_shifts
```

```
>     rep:=RepRadialshfs_Prod(parsed_ops,anorm,lambda,lambda_shfs,
>                                 nu_min-nu_min_shift,nu_max+nu_lap):

>     forget(RepRadial):
>     forget(RepRadial_param):
>     forget(RepRadialshfs_Prod):
>     forget(RepRadial_bS_DS):
>     forget(Matrix_sqrt):
>     forget(Matrix_sqrtInv):

>     # we need to return the matrix with index range nu_min..nu_max
>
>     if nu_lap=0 then
>       rep
>     else
>       SubMatrix(rep,1+nu_min_shift..1+nu_max-nu_min+nu_min_shift,
>                     1+nu_min_shift..1+nu_max-nu_min+nu_min_shift)
>     fi:

> end;


> # As above, but continues to remember everything.

> RepRadial_Prod_rem:=proc(rbs_op::list, anorm::algebraic,
>                          lambda::algebraic, lambda_var::integer,
>                          nu_min::nonnegint, nu_max::nonnegint,
>                          nu_lap::nonnegint:=0,$)
>     option remember;
>     local parsed_ops,lambda_shfs,nu_min_shift:

>     if evalf(lambda)<=0 then
>        error("Non-positive lambda value %1",lambda):
>     elif evalf(lambda+lambda_var)<=0 then
>        error("Non-positive lambda value %1",lambda+lambda_var):
>     fi:

>     # parse into a list of [K,T] and S ops (S=0,+1,-1)

>     parsed_ops:=Parse_RadialOp_List(rbs_op):

>     # assign lambda shifts to each of [K,T] and S.

>     lambda_shfs:=Lambda_RadialOp_List(parsed_ops,lambda_var):

>     # if resulting list is one larger than given, then need to
>     # prepend or append a [0,0] operator depending on whether
>     # the extra variation is positive or negative

>     if nops(lambda_shfs)>nops(parsed_ops) then
>       if lambda_shfs[1]>0 then
>         parsed_ops:=[ [0,0], op(parsed_ops) ]:
>       else
>         parsed_ops:=[ op(parsed_ops), [0,0] ]:
>         lambda_shfs:=[op(2..-1,lambda_shfs),lambda_shfs[1]]:
>       fi:
>     fi:

>     # now find representation of this product, with given
  lambda_shifts
>     # (we need to return the matrix with index range nu_min..
  nu_max)
```

```
>
>     if nu_lap=0 then
>       RepRadialshfs_Prod(parsed_ops,anorm,lambda,lambda_shfs,
>                                  nu_min,nu_max):
>     else
>       nu_min_shift:=min(nu_lap,nu_min):  # shift for lower index

>       SubMatrix(
>           RepRadialshfs_Prod(parsed_ops,anorm,lambda,lambda_shfs,
>                                  nu_min-nu_min_shift,nu_max+nu_lap),
>           1+nu_min_shift..1+nu_max-nu_min+nu_min_shift,
>           1+nu_min_shift..1+nu_max-nu_min+nu_min_shift)
>     fi:

> end;


> # The following parses a list of the basic radial operators
> #     Radial_b2, Radial_bm2, Radial_D2b, Radial_bDb,
> #     Radial_b,  Radial_bm, Radial_Db,
> #     Radial_S0, Radial_Sp, Radial_Sm,
> # (for beta^2; 1/beta^2; d^2/d(beta)^2; beta*d/d(beta);
> #  beta; 1/beta; d/d(beta); S0; S+; S-; respectively).

> # The return is a list of integers (-1,0 or 1) and pairs [K,T],
> # where the integers denote Radial_Sm, Radial_S0, Radial_Sp resp.,
> # and [K,T] denotes (beta^K * d/d(beta)^T).

> Parse_RadialOp_List:=proc(rs_op::list)

>   global Radial_Operators:
>   local i,T,K,POp_List,Pstate,idx:

>   POp_List:=[]:
>   T:=0:  # d^T/d(beta)^T
>   K:=0:  # (beta)^K

>   for i from nops(rs_op) to 1 by -1 do
>     if member(rs_op[i],[Radial_Sm,Radial_S0,Radial_Sp],'idx') then
>       if K<>0 or T>0 then
>         POp_List:=[ [K,T],op(POp_List) ]:   # write out [K,T]
>         K:=0:
>         T:=0:
>       fi:
>       POp_List:=[ idx-2,op(POp_List) ]:   # write out -1,0,1 for
  Sm,S0,Sp resp.
>     elif member(rs_op[i],[Radial_Db,Radial_D2b],'idx') then
>       if K<>0 then
>         POp_List:=[ [K,T],op(POp_List) ]:   # write out [K,T]
>         K:=0:
>         T:=0:
>       fi:
>       T:=T+idx:
>     elif rs_op[i]=Radial_bDb then
>       if K<>0 then
>         POp_List:=[ [K,T],op(POp_List) ]:   # write out [K,T]
>         T:=0:
>       fi:
>       T:=T+1:
>       K:=1:
>     elif member(rs_op[i],[Radial_b,Radial_b2],'idx') then
>       K:=K+idx:
```

```
>     elif member(rs_op[i],[Radial_bm,Radial_bm2],'idx') then
>       K:=K-idx:
>     else
>       error "operator %1 undefined", rs_op[i];
>     fi:
>   od:

>   if K<>0 or T>0 then
>     POp_List:=[ [K,T],op(POp_List) ]:   # write out any remaining
  [K,T]
>   fi:

>   POp_List:

> end:

> # Takes a list obtained from above, and assigns a lambda variation
> # to each term, so that we get the correct overall lambda change.
> # The elements of rsp_op are either integers (-1,0 or 1) or pairs
  [K,T].
> #
> # The returned list is usually the same size as that passed.
> # Otherwise, the returned list will be one longer, with the
> # first element corresponding to an extra [0,0] (i.e. identity op)
> # that should be prepended or appended to the list being passed.

> # This is implemented by partitioning lambda_var amongst the
  elements
> # of rsp_op, each of which has a nominal maximal value which is 0
> # for the integers (-1,0 or 1) and (|K|+T) for pairs [K,T].
> # We permit at most only one value exceeding the nominal maximal
> # value, and at most one that is of opposite parity.
> # The parity violation is in the leftmost possible position,
> # while the violation over the max is leftmost for lambda_var>0 and
> # rightmost for lambda_var<0.

> # Beware that having lambda variation on the Sm, S0, Sp operators
  is
> # open to confusion in that it doesn't change the lambda of the
> # SU(1,1) operators.


> Lambda_RadialOp_List:=proc(rsp_op::list,lambda_var::integer)
>   local i,n,var,oddin,max_vars,max_count,odd_vars,odd_count,
>         lambda_rem,lambda_list:

>   lambda_rem:=abs(lambda_var):
>   n:=nops(rsp_op):
>   lambda_list:=[0$n]:   # variations to be determined: initially
  all 0.
>   max_vars:=[0$n]:      # nominal maximal variations: determined in
  loop below

>   for i to n do
>     if type(rsp_op[i],list(integer)) then
>       max_vars[i]:=abs(rsp_op[i][1])+rsp_op[i][2]:
>     elif not type(rsp_op[i],integer) then
>       error "operator %1 undefined", rsp_op[i];
>     fi:
>   od:

>   max_count:=add(i,i in max_vars):
```

```
>   odd_vars:=map(irem,max_vars,2):  # take remainders mod 2
>   odd_count:=add(i,i in odd_vars): # number of odd entries
>   # set position of first odd (only needed for odd difference)
>   if type(lambda_rem-max_count,even) or not member(1,odd_vars,
  'oddin') then
>     oddin:=0
>   fi:

>   if lambda_rem<odd_count then
>     # set all odd entries to +/- 1 to get sum to be close to
  lambda_rem
>     lambda_list:=odd_vars:  # initially set all odd positions to 1
>     # then change first few to -1
>     for i to n while lambda_rem<odd_count do
>       if lambda_list[i]=1 then
>         lambda_list[i]:=-1:
>         odd_count:=odd_count-2:
>       fi:
>     od:

>     # if difference is odd, then need to set first odd position var
  to zero.

>     if oddin>0 then
>       lambda_list[oddin]:=0:
>     fi:

>     # entries in lambda_list here now add up to odd count.

>     # There is a small benefit in working from left to right above,
>     # in that if a d/d(beta) precedes a beta, they appear in rsp_op
>     # in separate terms. If lambda_var is 0, then the first gets
>     # -1 and the second +1. The resulting matrix for beta is
>     # then upper diagonal and the natural truncation in the product
>     # automatically gives the correct result.

>   elif lambda_rem<max_count then
>     lambda_list:=odd_vars:  # initially set all odd positions to 1
>     lambda_rem:=lambda_rem-odd_count:
>     for i to n while lambda_rem>0 do
>       # ensure we only add even values to each (perhaps 1 tooo
  much)
>       var:=2*iquo(min(lambda_rem+1,max_vars[i]-odd_vars[i]),2):
>       lambda_list[i]:=lambda_list[i]+var:
>       lambda_rem:=lambda_rem-var:
>     od:

>     if lambda_rem<0 then # 1 tooo many added, but no max is
  exceeded
>       if oddin>0 then
>         lambda_list[oddin]:=lambda_list[oddin]-1:
>       else # remove 1 from previous addition
>         lambda_list[i-1]:=lambda_list[i-1]-1:
>       fi
>     fi:

>   else
>     # put all values at maximum, with any excess going on the
>     # first, although if that excess is odd, increase it by 1
>     # and decrease the first odd maximum by 1.
>     lambda_list:=max_vars:
>     lambda_rem:=lambda_rem-max_count:
```

```
>     if oddin>0 then
>       # reduce first odd case by 1 (making lambda_rem even)
>       lambda_list[oddin]:=lambda_list[oddin]-1:
>       lambda_rem:=lambda_rem+1:
>     fi:
>     if lambda_rem>0 then # deal with excess lambda variation
>       if lambda_var>0 and n>0 and type(rsp_op[1],list(integer))
  then
>         # put remaining lambda_rem on first, which is [K,T]
>         lambda_list[1]:=lambda_list[1]+lambda_rem:
>       elif lambda_var<0 and n>0 and type(rsp_op[n],list(integer))
  then
>         # put remaining lambda_rem on last, which is [K,T]
>         lambda_list[n]:=lambda_list[n]+lambda_rem:
>       else # extend list at start (calling routine needs to test
  this)
>         lambda_list:=[lambda_rem,op(lambda_list)]:
>       fi:
>     fi:
>   fi:

>   if lambda_var>=0 then
>     lambda_list:
>   else
>     -lambda_list:
>   fi:

> end:


> # The following procedure is similar to RepRadial_Prod above, but
  is able to
> # represent linear combinations of products of the basic radial
  operators.
> # The arguments anorm, lambda, lambda_var, nu_min, nu_max are same
  as above,
> # the first, rlc_op, is of the form
> #         [ [coeff1,rs_op1], [coeff2,rs_op2], ...],
> # where rs_op1, rs_op2 are lists of basic radial operators, as in
  the first
> # argument above.
> # The return value is a Matrix, whose elements might need to be
  acted
> # upon by evalf to ensure that they are floats.

> # This procedure might eventually call any of the procedures
> #   RepRadial_Prod_rem, RepRadialshfs_Prod, RepRadial,
> #   RepRadial_param, Matrix_sqrt, Matrix_sqrtInv,
> # each of which uses a remember option.

> # We provide two versions, which differ only in that the first
  clears the
> # remember tables, while the second doesn't.
> # The first is the one described in the manual: it is not called by
> # anything in this code. The second is called only by the
  procedures
> # RepXspace_Pi, RepXspace_PiPi and RepXspace_PiqPi.

> RepRadial_LC:=proc(rlc_op::list(list), anorm::algebraic,
>                            lambda::algebraic, lambda_var::integer,
>                            nu_min::nonnegint, nu_max::nonnegint,
>                            nu_lap::nonnegint:=0)
```

```
>     local i,n,Mat;

>   n:=nops(rlc_op);

>   if n=0 then
>     Mat:=Matrix(nu_max-nu_min+1);  #Null matrix
>   else
>     Mat:=MatrixScalarMultiply(
> #            evalf(RepRadial_Prod_rem(rlc_op[1][2],anorm,
> #                                     lambda,lambda_var,nu_min,
  nu_max,nu_lap)),
>             RepRadial_Prod_rem(rlc_op[1][2],anorm,
>                                     lambda,lambda_var,nu_min,nu_max,
  nu_lap),
>             rlc_op[1][1]);
>     for i from 2 to n do
>       MatrixAdd(Mat,          # removed assignment (4/8/2015:
  unnecessary)
> #            evalf(RepRadial_Prod_rem(rlc_op[i][2],anorm,
> #                                     lambda,lambda_var,nu_min,
  nu_max,nu_lap)),
>             RepRadial_Prod_rem(rlc_op[i][2],anorm,
>                                     lambda,lambda_var,nu_min,nu_max,
  nu_lap),
>             1,rlc_op[i][1],inplace);
>     od:
>   fi:

>   # forget all possible remembered procedures this has called:

>   forget(RepRadial_Prod_rem):
>   forget(RepRadialshfs_Prod):
>   forget(RepRadial):
>   forget(RepRadial_param):
>   forget(Matrix_sqrt):
>   forget(Matrix_sqrtInv):
>
>   Mat;
> end:

> # As above, but everything is remembered.

> RepRadial_LC_rem:=proc(rlc_op::list(list), anorm::algebraic,
>                        lambda::algebraic, lambda_var::integer,
>                        nu_min::nonnegint, nu_max::nonnegint,
>                        nu_lap::nonnegint:=0)
>     option remember;
>     local i,n,Mat;

>   n:=nops(rlc_op);

>   if n=0 then
>     Mat:=Matrix(nu_max-nu_min+1);  #Null matrix
>   else
>     Mat:=MatrixScalarMultiply(
>             evalf(RepRadial_Prod_rem(rlc_op[1][2],anorm,
>                                     lambda,lambda_var,nu_min,nu_max,
  nu_lap)),
>             rlc_op[1][1]);
>     for i from 2 to n do
>       MatrixAdd(Mat,          # removed assignment (4/8/2015:
  unnecessary)
```

```
>             evalf(RepRadial_Prod_rem(rlc_op[i][2],anorm,
>                                      lambda,lambda_var,nu_min,nu_max,
  nu_lap)),
>             1,rlc_op[i][1],inplace);
>     od:
>   fi:
>
>   Mat;
> end:


> ##################################################################
  ########
> ####------------- Representations on the spherical space ----------
  ----####
> ##################################################################
  ########

> # The following two give the dimensions of SO(3) and SO(5)
> # irreducible representations (symmetric).

> dimSO3:=(L::nonnegint) -> 2*L+1:                    # SO(3) irrep
  dimension
> dimSO5:=(v::nonnegint) -> (v+1)*(v+2)*(2*v+3)/6:    # SO(5) irrep
  dimension

> # The following returns the total number of SO(3) irreps
> # (some possibly equivalent) in the SO(5) irrep of seniority v.

> dimSO5r3_allL:=(v::nonnegint) -> iquo( v*(v+3), 6 ) + 1:

> # The following sums the previous over the range v_min...v_max of
  seniorities.

> dimSO5r3_rngVallL:=(v_min::nonnegint,v_max::nonnegint)
>     -> add(dimSO5r3_allL(v),v=v_min..v_max):
>

> # The following procedure gives the multiplicity of SO(3) irreps of
> # angular momentum L in the SO(5) irrep of seniority v. It uses (6)
  .
> # It then provides the maximum value of the "missing" label alpha
> # (the minimum value is 1).

> dimSO5r3:=proc(v::integer,L::integer,$)
>   local b,d;

>   if v<0 or L<0 or L>2*v then
>     0:
>   else
>     b:=(L+3*irem(L,2))/2;
>     if v>=b then
>       d:=1+iquo(v-b,3);
>     else
>       d:=0;
>     fi:
>     if v>=L-2 then
>       d:=d-iquo(v-L+2,3);
>     fi:
>     d:
>   fi:
> end:
```

```
> # We now provide formulae similar to those above, counting SO(3)
  irreps,
> # but with L fixed or taking a range L_min,...,Lmax
> # (if Lmax is not given, then it is assumed that Lmin=Lmax).

> # The following counts SO(3) irreps for fixed v and a range of L.

> dimSO5r3_rngL:=(v::nonnegint,L_min::nonnegint,L_max::nonnegint)
>     -> add( dimSO5r3(v,j), j=L_min..L_max):

> # The following counts SO(3) irreps for a range of v and fixed L.

> dimSO5r3_rngV:=(v_min::nonnegint,v_max::nonnegint,L::nonnegint)
>     -> add( dimSO5r3(i,L),i=v_min..v_max):

> # The following counts SO(3) irreps for a range of v and a range of
  L.

> dimSO5r3_rngVrngL:=(v_min::nonnegint,v_max::nonnegint,
>                     L_min::nonnegint,L_max::nonnegint)
>     -> add(dimSO5r3_rngL(i,L_min,L_max),i=v_min..v_max):

> # The following also counts SO(3) irreps for a range of v and a
  range of L,
> # but if Lmax is not given, then it is assumed that Lmin=Lmax.

> dimSO5r3_rngVvarL:=(v_min::nonnegint,v_max::nonnegint,
>                     L_min::nonnegint,L_max::nonnegint)
>     -> `if`(_npassed>3,dimSO5r3_rngVrngL(_passed),
>                        dimSO5r3_rngV(_passed)):

> ##################################################################
  ########

> # We now specify procedures which give lists of labels that
  correspond
> # to the above counting/dimension formulae. In all but the first,
> # the labels are triples [v,alpha,L], where L gives the SO(3)
  angular
> # momentum and alpha is the "missing label" which distinguishes
> # SO(3) irreps having identical L and seniority v.
> # (Note that these are "reduced" labels for the states:
> #  the magnetic quantum label M is not included ---
> #  it would vary over the 2L+1 values  -L,-L+1,-L+2,...,L.)
> # These lists are used to label the states between which
  representation
> # matrices are constructed. In these lists of states, L varies
  slowest,
> # then v, with the missing label alpha varying quickest.

> # The following returns a list of labels [alpha,L] for all the
> # SO(3) irreps in a single SO(5) irrep of seniority v.

> lbsSO5r3_allL:=proc(v::nonnegint,$)
>     [seq(seq([a,LL],a=1..dimSO5r3(v,LL)),LL=0..2*v)]:
> end:


> # The following returns a list of labels [v,alpha,L] for the
> # range v_min...v_max of seniorities.

> lbsSO5r3_rngVallL:=proc(v_min::nonnegint,v_max::nonnegint,$)
```

```
>   if v_min>v_max then
>     error("Seniority range invalid");
>   else
>     [seq(seq(seq([u,a,LL],a=1..dimSO5r3(u,LL)),u=v_min..v_max),
>                                                    LL=0..2*
  v_max)]:
>   fi:
> end:

> # The following returns a list of labels [v,alpha,L] for a fixed
  seniority
> # v, but L restricted to the range L_min...L_max of SO(3) angular
  momenta.

> lbsSO5r3_rngL:=proc(v::nonnegint,L_min::nonnegint,L_max::nonnegint,
  $)
>   if L_min>L_max then
>     error("Parameter range invalid");
>   else
>     [seq(seq([v,a,LL],a=1..dimSO5r3(v,LL)),LL=L_min..L_max)]:
>   fi:
> end:

> # The following returns a list of labels [v,alpha,L] for a range of
> # seniorities v_min,..,vmax, but fixed SO(3) angular momentum L.

> lbsSO5r3_rngV:=proc(v_min::nonnegint,v_max::nonnegint,L::nonnegint,
  $)
>   if v_min>v_max then
>     error("Seniority range invalid");
>   else
>     [seq(seq([u,a,L],a=1..dimSO5r3(u,L)),u=v_min..v_max)]:
>   fi:
> end:

> # The following returns a list of labels [v,alpha,L] for ranges of
> # seniorities v_min,..,vmax, and SO(3) angular momenta L_min,..,
  Lmax.

> lbsSO5r3_rngVrngL:=proc(v_min::nonnegint,v_max::nonnegint,
>                         L_min::nonnegint,L_max::nonnegint,$)
>   if v_min>v_max or L_min>L_max then
>     error("Seniority range invalid");
>   else
>     [seq(seq(seq([u,a,LL],a=1..dimSO5r3(u,LL)),u=v_min..v_max),
>                                   LL=L_min..L_max)]:
>   fi:
> end:

> # The following returns a list of labels [v,alpha,L] for ranges of
> # seniorities v_min,..,vmax, and SO(3) angular momenta L_min,..,
  Lmax,
> # but the final argument L_max may be omitted, in which case L_max=
  L_min
> # (it may be used instead of the previous two procedures).

> lbsSO5r3_rngVvarL:=proc(v_min::nonnegint,v_max::nonnegint,
>                         L_min::nonnegint,L_max::nonnegint,$)
>   if v_min>v_max then
>     error("Seniority range invalid");
>   elif _npassed>3 then
>     [seq(seq(seq([u,a,LL],a=1..dimSO5r3(u,LL)),u=v_min..v_max),
```

```
>                                        LL=L_min..L_max)]:
>   else
>     [seq(seq([u,a,L_min],a=1..dimSO5r3(u,L_min)),u=v_min..v_max)]:
>   fi:
> end:


> ########################################################################
  ########
> #### SO(5) Clebsch-Gordon coefficients and reps of spherical
  harmonics ####
> ########################################################################
  ########

> # The files that contain the SO(5)>SO(3) CG coefficients
> # (v1,a1,L1,v2,a2,L2||v3,a3,L3) are assumed to lie below
> # the directory given in the global variable SO5CG_directory
> # (which must be set somewhere by the user).
> # The names of these files are of the form SO5CG_v1_v2-a2-L2_v3.
> # The are assumed to lie in directories named SO5CG_v1_v2_v3 which
> # themselves lie in directories named "v2=1/", "v2=2/", "v2=3/",
  etc.,
> # each a subdirectory of the directory specified in
  SO5CG_directory.
> # There are no v2=0 files because the data is easily generated
> # (and is done so in the procedure load_CG_table below).

> # The following procedure SO5CG_filename returns the full pathname
  of the
> # file that contains the SO(5)>SO(3) CG coefficients
> # (v1,a1,L1,v2,a2,L2||v3,a3,L3) for particular values of v1, v2,
  a2, L2, v3.

> SO5CG_filename:=proc(v1::nonnegint,
>                      v2::nonnegint,a2::posint,L2::nonnegint,
>                      v3::nonnegint)
>     cat(SO5CG_directory,"v2=",v2,"/SO5CG_",v1,"_",v2,"_",v3,
>     "/SO5CG_",v1,"_",v2,"-",a2,"-",L2,"_",v3);
> end:

>
> # The following procedure CG_labels returns, for the given v1,L2,
  v3,
> # a list of all quartets [a1,L1,a3,L3], where [v1,a1,L1] and [v3,
  a3,L3] are
> # valid SO(5)>SO(3) state labels, and for which |L1-L2| <= L3 <=
  L1+L2.
> # In this list L1 varies slowest, then a1 next slowest, then L3,
> # with a3 quickest.
> # The ordering (and length) of the list then accords with the order
  of the
> # SO(5)>SO(3) CG coefficients in the data files SO5CG_v1_v2-a2-
  L2_v3 .
> # The output of this routine is then used to correctly label the
  data
> # from the data file.
> # (Note that the data files contain all nine labels v1,a1,L1,v2,a2,
  L2,
> #  v3,a3,L3 alongside the CG coefficient, but these nine labels are
> #  not read - they are assumed to accord with the labels given by
> #  CG_labels (we could thus make the data files smaller).)
```

```
> # (Note that the list returned here is generally smaller than the
> #  direct product of the sets obtained from calling
> #  lbsSO5r3_allL(v1) and lbsSO5r3_allL(v3).)

> CG_labels:=proc(v1::nonnegint,
>                 L2::nonnegint,
>                 v3::nonnegint)
>   local L1,L3,a1,a3,label_list;
>   label_list:=[]:

>   for L1 from 0 to 2*v1 do
>   for a1 to dimSO5r3(v1,L1) do
>     for L3 from abs(L1-L2) to min(L1+L2,2*v3) do
>     for a3 to dimSO5r3(v3,L3) do
>         label_list:=[op(label_list),[a1,L1,a3,L3]]:
>     od: od:
>   od: od:

>   label_list:
> end:

> # The following procedure get_CG_file reads the
> # SO(5)>SO(3) CG coefficients from the data file SO5CG_v1_v2-a2-
  L2_v3 .
> # It returns a list of two values, the first of which is a list
> # of floats giving the coefficients, and the second of which is
> # the list of corresponding labels [a1,L1,a3,L3], the latter
  obtained
> # using the procedure CG_labels above.
> # It is only used by the subsequent procedure.
>
> get_CG_file:=proc(v1::nonnegint,
>                   v2::nonnegint,a2::posint,L2::nonnegint,
>                   v3::nonnegint)
>   local CG_data,CG_list:

>   if v1>v2+v3 or v2>v3+v1 or v3>v1+v2 or type(v1+v2+v3,odd)
>       or v3<v1         # data is obtained from v3>v1 cases
>       or a2>dimSO5r3(v2,L2) then
>     error "No CG file for these parameters!":
>   fi:

>   CG_data:=readdata( SO5CG_filename(v1,v2,a2,L2,v3), float):
>   CG_list:=CG_labels(v1,L2,v3):
>   [CG_data,CG_list]:
> end:

> # The following procedure show_CG_file displays the
> # SO(5)>SO(3) CG coefficients from the data file SO5CG_v1_v2-a2-
  L2_v3 .
> # For each label [a1,v1,a3,L3], it prints the label followed by the
> # value of the corresponding CG coefficient.
> # The procedure is useful for testing that the CG coefficients are
  being
> # accessed correctly. It is not used subsequently.
>
> show_CG_file:=proc(v1::nonnegint,
>                    v2::nonnegint,a2::posint,L2::nonnegint,
>                    v3::nonnegint)
>   local cg_table,count,i:

>   cg_table:=get_CG_file(v1,v2,a2,L2,v3):
```

```
>   count:=nops(cg_table[1]):
>   if count=1 then
>     print(`This file contains ` || count || ` CG coefficient`):
>   else
>     print(`This file contains ` || count || ` CG coefficients`):
>   fi:

>   for i to count do
>     print(cg_table[2][i],cg_table[1][i]):
>   od:
> end:


> # The following procedure load_CG_table loads all the
> # SO(5)>SO(3) CG coefficients for a particular (v1,v2,a2,L2,v3)
> # from the data file SO5CG_v1_v2-a2-L2_v3 .
> # They are loaded into the table CG_coeffs (which was initialised
  above),
> # from where they can be readily accessed.
> # Subsequent attemps to load the same data will be silently
  ignored.
> # Note that no checking is done here on the correct ranges of the
> # arguments (this is left to the functions that call this).
> # We should restrict to
> #   v1<=v2+v3 and v2<=v3+v1 and v3<=v1+v2 and type(v1+v2+v3,odd)
> #      and a2<=dimSO5r3(v2,L2).
> # We should also restrict to v values at most the maximal seniority
> # of that of the data files (currently 6), else a "file does not
  exist"
> # error will be generated.

> # Note that if v1>v3, then the data for (v3,v2,a2,L2,v1) is loaded
  instead
> # because the SO(5)>SO(3) CG coefficients for (v1,v2,a2,L2,v3) are
  easily
> # obtained from the former using (4.164) of [RowanWood].
> # Also note that for v2=0, no data is read, but each coefficient
> # is set correctly to 1.0.

> load_CG_table:=proc(v1::nonnegint,
>                     v2::nonnegint,a2::posint,L2::nonnegint,
>                     v3::nonnegint)
>   local CG_data,CG_list,vt1,vt3;
>   global CG_coeffs;

>   if v1>v3 then
>     vt1:=v3: vt3:=v1:
>   else
>     vt1:=v1: vt3:=v3:
>   fi:
>   if evalb([vt1,v2,a2,L2,vt3] in [indices(CG_coeffs)] ) then
>     return:
>   fi:

>   CG_list:=CG_labels(vt1,L2,vt3):
>   if v2>0 then  # read data from file
>     CG_data:=readdata( SO5CG_filename(vt1,v2,a2,L2,vt3), float):
>   else  # generate data
>     CG_data:=[1.0$nops(CG_list)]:
>   fi:

>   CG_coeffs[vt1,v2,a2,L2,vt3]:=table([seq( (op(CG_list[i]))=CG_data
```

```
  [i],
>                                   i=1..nops(CG_list) )]);
> end:

> # The following procedure CG_SO5r3 returns the SO(5)>SO(3) CG
  coefficient
> # (v1,a1,L1;v2,a2,L2||v3,a3,L3)  [no renormalisation required].
> # The return value is a float.
> # If not already present in CG_coeffs, the data file for
> # (v1,v2,a2,L2,v3) is loaded.
> # Note that if v1>v3, then the data for (v3,v2,a2,L2,v1) is used
  instead,
> # and a factor is included (see (4.164) of [RowanWood]).
> # (A faster version that does no testing of indices could be
  written).

> CG_SO5r3:=proc(v1::nonnegint,a1::posint,L1::nonnegint,
>                v2::nonnegint,a2::posint,L2::nonnegint,
>                v3::nonnegint,a3::posint,L3::nonnegint)
>   global CG_coeffs;
>   if v1+v2<v3 or v1+v3<v2 or v2+v3<v1 or
>      L1+L2<L3 or L1+L3<L2 or L2+L3<L1 or
>      a1>dimSO5r3(v1,L1) or a2>dimSO5r3(v2,L2)
>                         or a3>dimSO5r3(v3,L3) or type(v1+v2+v3,odd)
  then
>         0;
>   else
>      load_CG_table(v1,v2,a2,L2,v3);
>      if v1<=v3 then
>        CG_coeffs[v1,v2,a2,L2,v3][a1,L1,a3,L3]:
>      else
>        CG_coeffs[v3,v2,a2,L2,v1][a3,L3,a1,L1]
>          * (-1)^(L3+L2-L1)
>          * sqrt( dimSO5(v3) * dimSO3(L1) / dimSO5(v1) / dimSO3(L3)
  ):
>      fi:
>   fi:
> end:


> ##################################################################
  ########

> # The following procedure CG_SO3 returns the usual SO(3) Clebsch-
  Gordon
> # coefficients CG(j1,m1,j2,m2;j3,m3). Here the arguments are each
> # 1/2 integers (rationals). The return value is expressed as a
  surd.
> # The formula used is that of eqn. (3.6.10) of Edmond's book.

> # In the ACM code, this procedure is only used, in some instances,
> # to calculate transition amplitudes from transition rates.
> # It's not used elsewhere.

> # We could use it, for example, via (36) to get full matrix
> # elements of spherical harmonics Y^v_{aLM}, by
> #   CG_SO3(L_i,M_i,0,0,L_f,M_f)*ME_SO5r3(v_f,al_f,L_f,v,a,L,,v_i,
  al_i,L_i):

> CG_SO3:=proc(j1::rational,m1::rational,j2::rational,m2::rational,
>                                        j3::rational,m3::rational)
```

```
>   if abs(m1)>j1 or abs(m2)>j2 or abs(m3)>j3 then RETURN (0) fi;
>   if not(type(j1+m1,integer) and type(j2+m2,integer)
>                                and type (j3+m3,integer)) then
  RETURN (0) fi;
>   if m1+m2 <> m3 then RETURN (0) fi;
>   if j3 < abs(j1-j2) or j3 > j1+j2 then RETURN (0) fi;

>   (-1)^(j1-j2+m3)*
>       simplify(Wigner_3j(j1,j2,j3,m1,m2,-m3)*sqrt(2*j3+1),sqrt);
> end:


> Wigner_3j:=proc(j1,j2,j3,m1,m2,m3)
>      (-1)^(2*j1-j2+m2)
>        * sqrt((j1+j2-j3)!*(j1-m1)!*(j2-m2)!*(j3-m3)!*(j3+m3)!/
>               ((j1+j2+j3+1)!*(j1-j2+j3)!*(-j1+j2+j3)!*(j1+m1)!*(j2+
  m2)!))
>        * add((-1)^s*(j1+m1+s)!*(j2+j3-m1-s)!/
>                    (s!*(j1-m1-s)!*(j2-j3+m1+s)!*(j3+m3-s)!),
>              s=max(0,j3-j2-m1)..min(j3+m3,j1-m1))
> end:



> ########################################################################
  ########
> ################## Representations of spherical harmonics
  ##################
> ########################################################################
  ########

> # The following procedure ME_SO5red returns the doubly reduced
  matrix
> # element <u|||w|||v>*4*Pi for SO(5) spherical harmonic tensor
  operators.
> # It uses (45).
> # The return value is algebraic and exact (and probably a surd).

> ME_SO5red:=proc(u::nonnegint,w::nonnegint,v::nonnegint)
>   local sigma,halfsigma;
>   if u+v<w or u+w<v or v+w<u or type(u+v+w,odd) then
>     RETURN(0);
>   fi:

>   sigma:=(v+w+u); halfsigma:=sigma/2;
>   (halfsigma+1)! / (halfsigma-u)! / (halfsigma-v)! / (halfsigma-w)!
>     * sqrt( (2*v+3) * (2*w+3) * (sigma+4) / (u+2) / (u+1)
>             * (sigma-2*u+1)! * (sigma-2*w+1)! * (sigma-2*v+1)! /
  (sigma+3)! );
> end:


> # The following nine functions are useful instances of the above,
> # with different normalisations: they provide SO(5) (doubly)
  reduced
> # matrix elements for Q and [QxQ]_(v=2) and [QxQxQ]_(v=3).
> # They may be obtained from ME_SO5red by using
> #     Q=4*Pi/sqrt(15) * Y_{112},
> #     [QxQ]^2_4=4*Pi*sqrt(2/105) * Y_{214}, and
> #     [QxQxQ]^3_6=4*Pi*sqrt(2/315) * Y_{316}
> # (we need to take care with different signs for
> #       [QxQ]^2_2=-4*Pi*sqrt(2/105) * Y_{212} and
```

```
> #       [QxQxQ]^3_0=-4*Pi*sqrt(2/315) * Y_{310} ).

> # The following gives <v+1|||Q|||v> and <v-1|||Q|||v>

> Qred_p1:=(v) -> sqrt((v+1)/(2*v+5)):
> Qred_m1:=(v) -> sqrt((v+2)/(2*v+1)):

> # The following gives <v+2|||QxQ|||v>, <v|||QxQ|||v> &
  <v-2|||QxQ|||v>

> QxQred_p2:=(v) -> sqrt((v+1)*(v+2)/(2*v+5)/(2*v+7)):
> QxQred_0:=(v)  -> sqrt(6*v*(v+3)/5/(2*v+1)/(2*v+5)):
> QxQred_m2:=(v) -> sqrt((v+1)*(v+2)/(2*v+1)/(2*v-1)):

> # The following gives <v+3|||QxQxQ|||v>, <v+1|||QxQxQ|||v>
> #                     <v-1|||QxQxQ|||v>, <v-3|||QxQxQ|||v>

> QxQxQred_p3:=(v) -> sqrt((v+1)*(v+2)*(v+3)/(2*v+5)/(2*v+7)/(2*v+9))
  :
> QxQxQred_p1:=(v) -> 3*sqrt(v*(v+1)*(v+4)/7/(2*v+1)/(2*v+5)/(2*v+7))
  :
> QxQxQred_m1:=(v) -> 3*sqrt((v-1)*(v+2)*(v+3)/7/(2*v-1)/(2*v+1)/(2*
  v+5)):
> QxQxQred_m3:=(v) -> sqrt(v*(v+1)*(v+2)/(2*v-3)/(2*v-1)/(2*v+1)):


> # The following procedure ME_SO5r3 returns the (alternative SO(3)
  reduced)
> # matrix element
> #
> #       4*Pi
> #    ------------- * <v_f,al_f,L_f || Y^v_{al,L} || v_i,al_i,L_i>
> #    sqrt (2*L_f+1)
> #
> # for the SO(5) spherical harmonic Y^v_{al,L}. It uses (37) & (39).
> # The return value might be (partly) algebraic, so it might be
> # necessary to apply evalf to it.

> # To obtain alternative reduced matrix elements of cos(3g), use
> #               Y310 = (3/4/Pi) cos(3g).
> # So to get the matrix element of cos(3g) from this, we must mult
  by (1/3).

> ME_SO5r3:=proc(v_f::integer,al_f::integer,L_f::integer,
>                v::integer,al::integer,L::integer,
>                v_i::integer,al_i::integer,L_i::integer)

>    CG_SO5r3(v_i,al_i,L_i,v,al,L,v_f,al_f,L_f) * ME_SO5red(v_f,v,
  v_i):
> end;


> # The following procedure RepSO5_Y_rem returns a Matrix of
> # (alternative SO(3) reduced) matrix elements
> #
> #       4*Pi
> #    ------------- * <v_f,al_f,L_f || Y^v_{al,L} || v_i,al_i,L_i>
> #    sqrt (2*L_f+1)
> #
> # for v_min <= v_i,v_f <= v_max and L_min <= L_i,L_f <= L_max.
> # It thus provides a representation of the spherical harmonic
> # Y^v_{al,L} (but lacking sqrt(2L_f+1)/4/Pi factors).
```

```
> # The elements of the returned Matrix are all floats (evalf used
  within).

> # If the L_max argument is omitted, then L_max=L_min, and thus
> # a single value of angular momentum is used.
> # If [v,al,L] doesn't label a spherical harmonic, then a matrix of
  0s is
> # returned.

> # The remember option is used, but these stored values are cleared
> # each time the (much later) RepXspace() routine is invoked.

> RepSO5_Y_rem:=proc(v::integer,al::integer,L::integer,
>                    v_min::integer,v_max::integer,
>                    L_min::integer,L_max::integer,$)
>   option remember;
>   local states:

>   states:=lbsSO5r3_rngVvarL(_passed[4..-1]):
>   Matrix( nops(states), (i,j)->evalf(
>                 ME_SO5r3(op(states[i]),v,al,L,op(states[j])) )):
> end:


> # The following procedure RepSO5_Y_alg is the same as RepSO5_Y_rem
> # except that evalf is not used, and thus the elements of the
> # returned matrix come out (partially) algebraic.
> # Also, the remember option is not used.
> # This procedure is not used elsewhere.

> RepSO5_Y_alg:=proc(v::integer,al::integer,L::integer,
>                    v_min::integer,v_max::integer,
>                    L_min::integer,L_max::integer,$)
>   local states:

>   states:=lbsSO5r3_rngVvarL(_passed[4..-1]):
>   Matrix( nops(states), (i,j)->
>                 ME_SO5r3(op(states[i]),v,al,L,op(states[j])) ):
> end:



> # The following procedure RepSO5_sqLdim returns a Matrix acting on
> # the states with v_min <= v_i,v_f <= v_max and L_min <= L_i,L_f <=
  L_max,
> # which is diagonal with entries (-1)^L_i*sqrt(2L_i+1).

> RepSO5_sqLdim:=proc(v_min::integer,v_max::integer,
>                    L_min::integer,L_max::integer,$)
>   local states:

>   states:=lbsSO5r3_rngVvarL(_passed):  # obtain labels for states
>   Matrix(map(x->evalf(eval((-1)^(x[3])*sqrt(dimSO3(x[3]))))),
  states),
>                                shape=diagonal,scan=diagonal);
> end:

> # The following procedure RepSO5_sqLdiv returns a Matrix acting on
> # the states with v_min <= v_i,v_f <= v_max and L_min <= L_i,L_f <=
  L_max,
> # which is diagonal with entries (-1)^L_i/sqrt(2L_i+1).
```

```
> RepSO5_sqLdiv:=proc(v_min::integer,v_max::integer,
>                     L_min::integer,L_max::integer,$)
>   local states:

>   states:=lbsSO5r3_rngVvarL(_passed):  # obtain labels for states
>   Matrix(map(x->evalf(eval((-1)^(x[3])/sqrt(dimSO3(x[3])))),
  states),
>                                shape=diagonal,scan=diagonal);
> end:


> # The following procedure RepSO5r3_Prod returns a Matrix that
  represents
> # (up to a normalisation given below) the product of spherical
  harmonics
> # on the space of states with v_min <= v_i,v_f <= v_max and
> # L_min <= L_i,L_f <= L_max. The Matrix has entries of type float.
> # If the L_max argument is omitted, then L_max=L_min.
> # The argument ys_op is a list, each element of which denotes
> # a single spherical harmonic. Each of these elements is either one
  of
> # the symbolic names
> #     SpHarm_112, SpHarm_212, SpHarm_214, ... , SpHarm_61C
> # (see SpHarm_Table above for the full list),
> # or is a triple [v,alpha,L] which designates the Spherical
  harmonic
> # explicitly. In addition, two other operators are also permitted:
  they are
> #   1) SpDiag_sqLdim, which provides a diagonal operator with
  entries
> #                          (-1)^{L_i}*sqrt(2L_i+1)
> #   2) SpDiag_sqLdiv, which provides a diagonal operator with
  entries
> #                          (-1)^{L_i}/sqrt(2L_i+1).

> # The returned Matrix is obtained simply by multiplying
> # together the individual Matrixes for the entries of ys_op.
> # Therefore, the result is meaningful only for certain products:
> #   1. at most one entry having non-zero angular momentum;
> #   2. Same, but also with products of the form
> #            [..., SpDiag_sqLdim, Y1, SpDiag_sqLdiv, Y2, ...]
> #      with Y1 and Y2 having identical AM.
> # In these cases, on multiplying the returned Matrix by (4*Pi)^(-
  N),
> # where N is the number of genuine spherical harmonics in the list,
> # the result is a Matrix of alternative SO(3)-reduced matrix
  elements.
> # (N may be obtained using the procedure NumSO5r3_Prod below,
> # and is not necessarily the length of ys_op because this list may
> # contain non-harmonics such as SpDiag terms).
> # To obtain genuine SO(3)-reduced matrix elements, the result
> # needs to be further multiplied by (a matrix of) sqrt(2L_f+1).

> # This procedure clears the RepSO5_Y_rem remember tables.

> RepSO5r3_Prod:=proc(ys_op::list,v_min::integer,v_max::integer,
>                                  L_min::integer,L_max::integer,$)
>   local rep:
>   rep:=RepSO5r3_Prod_wrk(_passed):

>   forget(RepSO5_Y_rem):
>   rep:
```

```
> end:

> # The following procedure RepSO5r3_Prod_rem is exactly the same as
> # the above except that has the remember option, and doesn't clear
> # the remember tables for RepSO5_Y_rem.

> RepSO5r3_Prod_rem:=proc(ys_op::list,v_min::integer,v_max::integer,
>                                     L_min::integer,L_max::integer,
  $)
>     option remember;

>   RepSO5r3_Prod_wrk(_passed):
> end:

> # The following procedure RepSO5r3_Prod_wrk is as the above two,
> # but does all the work for those, without being concerned by
> # remembering stuff.
> # Note the use of "copy" in this procedure. This is necessary
  otherwise
> # the remember table for RepSO5_Y_rem gets messed up!

> RepSO5r3_Prod_wrk:=proc(ys_op::list,v_min::integer,v_max::integer,
>                                     L_min::integer,L_max::integer,
  $)
>     local i,n,Mats,Mat_product,this_op;
>     global SpHarm_Operators,SpHarm_Table;

>   n:=nops(ys_op);

>   if n=0 then   # require identity matrix
>     return Matrix([seq(1,i=1..dimSO5r3_rngVvarL(_passed[2..-1]))],
>                                     scan=diagonal):
>   else

>     for i from 1 to n do

>       if type(ys_op[i],list(nonnegint)) and nops(ys_op[i])=3 then
>         this_op:=ys_op[i];
>       elif member(ys_op[i],SpHarm_Operators) then
>         this_op:=SpHarm_Table[ys_op[i]];
>       elif ys_op[i]=SpDiag_sqLdim then
>         if i=1 then  # cannot make copy because that'd force diag
  data matrix
>           Mat_product:=Matrix(RepSO5_sqLdim(_passed[2..-1]));
>         else
>           MatrixMatrixMultiply(Mat_product,
>                                RepSO5_sqLdim(_passed[2..-1]),
  inplace);
>         fi:
>         next;     # tackle next i in for loop
>       elif ys_op[i]=SpDiag_sqLdiv then
>         if i=1 then
>           Mat_product:=Matrix(RepSO5_sqLdiv(_passed[2..-1]));
>         else
>           MatrixMatrixMultiply(Mat_product,
>                                RepSO5_sqLdiv(_passed[2..-1]),
  inplace);
>         fi:
>         next;     # tackle next i in for loop

>       else
>         error "Invalid SO(5) harmonic designator %1", ys_op[i]:
```

```
>       fi:


>       # Now multiply in the spherical harmonic denoted by this_op.

>       if i=1 then
>         Mat_product:=copy(RepSO5_Y_rem( op(this_op), _passed[2..-1]
  ));
>       else
>         MatrixMatrixMultiply(
>                    Mat_product,
>                    RepSO5_Y_rem( op(this_op), _passed[2..-1]),
>                    inplace);
>       fi:

>     od:
>   fi:
>
> #  To get genuine alternative reduced matrix elements, we now need
  to
> #  multiply by (4*Pi)^(-N) for N the number of SpHarms in the list

>    Mat_product;
> end:


> # The following procedure NumSO5r3_Prod examines the list ys_op,
  and
> # determines how many of its entries denote spherical harmonics,
> # either from
> #     SpHarm_112, SpHarm_212, SpHarm_214, ... , SpHarm_61C
> # (see SpHarm_Table above for the full list),
> # or a triple [v,alpha,L] (but no checking that the entries
  actually
> # correspond to a genuine spherical harmonic).

> # This is useful for getting the correct 4*Pi normalisation in the
> # previous few procedures.

> NumSO5r3_Prod:=proc(ys_op::list,$)
>   local i,ct:

>   ct:=0:

>   for i from 1 to nops(ys_op) do
>     if type(ys_op[i],list(nonnegint)) and nops(ys_op[i])=3 then
>         ct:=ct+1:
>     elif member(ys_op[i],SpHarm_Operators) then
>         ct:=ct+1:
>     fi:
>   od:

>   ct;
> end:


> ##################################################################
  ########
> ####------------------ Specification of Operators ---------------
  ----####
> ##################################################################
  ########
```

```
> # In the ACM, operators on the full Hilbert space are encoded using
> # a particular list structure. These lists are each of the form
> #      [ [co1, [op11,op12,...] ],
> #        [co2, [op21,op22,...] ],
> #        [co3, [op31,op32,...] ],
> #        ...
> #        [coN, [opN1,opN2,...] ] ]
> # where each co# is a constant, and each op## is a symbolic name
> # from table I,II or III, or SpDiag_sqLdim or SpDiag_sqLdiv.
> # This then corresponds to the operator which is the sum of
  operators
> #                co# * Op#1 * Op#2 * Op#3 * ...
> # where Op#i is the operator denoted by the symbolic name op#i,
> # and SpDiag_sqLdim and SpDiag_sqLdiv are diagonal operators that
> # multiply the basis state [nu;v,alpha,L] by
> #            (-1)^L*sqrt(2L+1) and (-1)^L/sqrt(2L+1)
> # respectively.
> # (The full list of acceptable symbolic names is given in the
  variables
> #  Radial_Operators, Spherical_Operators, Xspace_Operators.)

> # In addition to simple numerical constants, the coefficients
> # can be functions of SENIORITY, ANGMOM, NUMBER, ALFA
> # (anything else will cause problems!) -
> # these will be substituted for according to the [nu;v,alpha,L]
  values
> # of the state being operated on by setting SENIORITY=v, ANGMOM=L,
> # NUMBER=nu, ALFA=alpha.

> # See Section 7.3 for more details.

> ##################################################################
  ########

> # The procedure ACM_Hamiltonian below produces the encoding of
> # operators of certain (rational) types. Thus for these operators,
> # the user doesn't need to know anything about the encoding method.
> # This procedure takes up to 14 parameters that specify
  coefficients
> # of (b denotes beta, g denotes gamma)
> #    Laplacian, 1, b^2, b^4,b^(-2),
> #      b*cos(3g), b^3*cos(3g), b^5*cos(3g), b^(-1)*cos(3g),
> #      cos(3g)^2, b^2*cos(3g)^2, b^4*cos(3g)^2, b^(-2)*cos(3g)^2,
> #      [pi x q x pi]_(v=3,L=0).
> # Each of the arguments is a numeric value, or a function of
> # SENIORITY, ANGMOM, NUMBER, ALFA, as described above.

> ACM_Hamiltonian:=proc(c11:=0,c20:=0,c21:=0,c22:=0,c23:=0,
>                           c30:=0,c31:=0,c32:=0,c33:=0,
>                           c40:=0,c41:=0,c42:=0,c43:=0,
>                           c50:=0,$)
>     local our_op:

>   if c11<>0 then   # build laplacian using eqn (57)
>     our_op:=[ [c11,[Radial_D2b]],
>               [-c11*(2+SENIORITY*(SENIORITY+3)),[Radial_bm2]] ]:
>   else
>     our_op:=[]:
>   fi:

>   if c20<>0 then our_op:=[ op(our_op),
>                 [c20,[]] ]: fi:
```

```
>   if c21<>0 then our_op:=[ op(our_op),
>                [c21,[Radial_b2]] ]: fi:
>   if c22<>0 then our_op:=[ op(our_op),
>                [c22,[Radial_b2,Radial_b2]] ]: fi:
>   if c23<>0 then our_op:=[ op(our_op),
>                [c23,[Radial_bm2]] ]: fi:
>   if c30<>0 then our_op:=[ op(our_op),
>                [c30*Convert_310,[Radial_b,SpHarm_310]] ]: fi:
>   if c31<>0 then our_op:=[ op(our_op),
>                [c31*Convert_310, [Radial_b2,Radial_b,SpHarm_310]] ]
  : fi:
>   if c32<>0 then our_op:=[ op(our_op),
>                [c32*Convert_310, [Radial_b2,Radial_b2,
>                                   Radial_b,SpHarm_310]] ]: fi:
>   if c33<>0 then our_op:=[ op(our_op),
>                [c33*Convert_310, [Radial_bm,SpHarm_310]] ]: fi:
>   if c40<>0 then our_op:=[ op(our_op),
>                [c40*Convert_310^2, [SpHarm_310,SpHarm_310]] ]: fi:
>   if c41<>0 then our_op:=[ op(our_op),
>                [c41*Convert_310^2, [Radial_b2,SpHarm_310,
  SpHarm_310]] ]: fi:
>   if c42<>0 then our_op:=[ op(our_op),
>                [c42*Convert_310^2, [Radial_b2,Radial_b2,
>                                     SpHarm_310,SpHarm_310]] ]: fi:
>   if c43<>0 then our_op:=[ op(our_op),
>                [c43*Convert_310^2,[Radial_bm2,SpHarm_310,
  SpHarm_310]] ]: fi:
>   if c50<>0 then our_op:=[ op(our_op),
>                [c50,[Xspace_PiqPi]] ]: fi:

>   our_op:
> end:

> # The procedure ACM_HamRigidBeta below produces the encoding of
> # certain Hamiltonians that are appropriate for rigid-beta models
> # (they don't involve beta). There are up to eight numerical
> # arguments that stipule the coefficients of
> #    SO(5) Casimir, 1, cos(3g)
> #    cos(3g)^2, cos(3g)^3, cos(3g)^4, cos(3g)^5, cos(3g)^6.
> # The final argument (0 or 1, the former the default) indicates
  whether
> # to encode using only the spherical harmonic SpHarm_310 (for 0),
  or use
> # the spherical harmonic SpHarm_610 as much as possible (for 1).

> ACM_HamRigidBeta:=proc(cas:=0,con:=0,c1:=0,c2:=0,c3:=0,
>                                     c4:=0,c5:=0,c6:=0,
  flag::integer:=0,$)
>     local our_op:


>   if flag=0 then
>     our_op:=ACM_HamSH3(_params[2..8]):
>   elif flag=1 then
>     our_op:=ACM_HamSH6(_params[2..8]):
>   else
>     error "Unrecognised flag %1", flag:
>   fi:

>   if cas<>0 then   # build casimir using eqn (58)
>     if our_op<>[] then
>       our_op:=[ [cas*SENIORITY*(SENIORITY+3),[]], op(our_op) ]:
```

```
>     else
>       our_op:=[ [cas*SENIORITY*(SENIORITY+3),[]] ]:
>     fi:
>   fi:

>   our_op:
> end:

> # The procedure ACM_HamSH3 below provides the ACM encoding for
  linear
> # combinations of
> #    1, cos(3g), cos(3g)^2, cos(3g)^3, cos(3g)^4, cos(3g)^5, cos
  (3g)^6,
> #    cos(3g)^7, cos(3g)^8,
> # in terms of the spherical harmonic SpHarm_310.
> # Its nine arguments give the coefficients of these terms.
> # It's used by the above procedure ACM_HamRigidBeta.

> ACM_HamSH3:=proc(c0:=0,c1:=0,c2:=0,c3:=0,c4:=0,c5:=0,c6:=0,c7:=0,
  c8:=0,$)
>     local our_op:

>   if c0<>0 then
>       our_op:=[ [c0,[]] ]:
>   else
>       our_op:=[]:
>   fi:


>   if c1<>0 then our_op:=[ op(our_op),
>               [c1*Convert_310,[SpHarm_310]] ]: fi:
>   if c2<>0 then our_op:=[ op(our_op),
>               [c2*Convert_310^2,[SpHarm_310,SpHarm_310]] ]: fi:
>   if c3<>0 then our_op:=[ op(our_op),
>               [c3*Convert_310^3,[SpHarm_310,SpHarm_310,SpHarm_310]]
  ]: fi:
>   if c4<>0 then our_op:=[ op(our_op),
>               [c4*Convert_310^4,[SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310]] ]: fi:
>   if c5<>0 then our_op:=[ op(our_op),
>               [c5*Convert_310^5,[SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310,SpHarm_310]] ]: fi:
>   if c6<>0 then our_op:=[ op(our_op),
>               [c6*Convert_310^6,[SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310,SpHarm_310,SpHarm_310]]
  ]: fi:
>   if c7<>0 then our_op:=[ op(our_op),
>               [c7*Convert_310^7,[SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310]] ]: fi:
>   if c8<>0 then our_op:=[ op(our_op),
>               [c8*Convert_310^8,[SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310,SpHarm_310,SpHarm_310,
>                                  SpHarm_310,SpHarm_310]] ]: fi:
>   our_op:
> end:


> # The procedure ACM_HamSH6 below provides the ACM encoding for
  linear
> # combinations of
> #    1, cos(3g), cos(3g)^2, cos(3g)^3, cos(3g)^4, cos(3g)^5, cos
```

```
  (3g)^6,
> #    cos(3g)^7, cos(3g)^8,
> # in terms of the spherical harmonics SpHarm_310 and SpHarm_610,
> # preferring the latter as much as possible.
> # Its nine arguments give the coefficients of these terms.
> # It's used by the above procedure ACM_HamRigidBeta.

> ACM_HamSH6:=proc(c0:=0,c1:=0,c2:=0,c3:=0,c4:=0,c5:=0,c6:=0,c7:=0,
  c8:=0,$)
>     local our_op,d0,d1,d2,d3,d4,d5,d6,d7,d8:

>   # First convert into coefficients d[2n+m] of
>   #   (Convert_310*SpHarm_310)^m * (Convert_610*SpHarm_610)^n
>   # where m=0 or 1, and n>=0.

>   d0:=c0+c2/3+c4/9+c6/27+c8/81:
>   d1:=c1+c3/3+c5/9+c7/27:
>   d2:=c2/3+c4*2/9+c6/9+c8*4/81:
>   d3:=c3/3+c5*2/9+c7/9:
>   d4:=c4/9+c6/9+c8*2/27:
>   d5:=c5/9+c7/9:
>   d6:=c6/27+c8*4/81:
>   d7:=c7/27:
>   d8:=c8/81:


>   if d0<>0 then
>        our_op:=[ [d0,[]] ]:
>   else
>        our_op:=[]:
>   fi:

>   if d1<>0 then our_op:=[ op(our_op),
>               [d1*Convert_310, [SpHarm_310]] ]: fi:
>   if d2<>0 then our_op:=[ op(our_op),
>               [d2*Convert_610, [SpHarm_610]] ]: fi:
>   if d3<>0 then our_op:=[ op(our_op),
>               [d3*Convert_610*Convert_310, [SpHarm_610,SpHarm_310]]
  ]: fi:
>   if d4<>0 then our_op:=[ op(our_op),
>               [d4*Convert_610^2, [SpHarm_610,SpHarm_610]] ]: fi:
>   if d5<>0 then our_op:=[ op(our_op),
>               [d5*Convert_610^2*Convert_310,
>                           [SpHarm_610,SpHarm_610,SpHarm_310]] ]:
  fi:
>   if d6<>0 then our_op:=[ op(our_op),
>               [d6*Convert_610^3,
>                           [SpHarm_610,SpHarm_610,SpHarm_610]] ]:
  fi:
>   if d7<>0 then our_op:=[ op(our_op),
>               [d7*Convert_610^3*Convert_310,
>                           [SpHarm_610,SpHarm_610,SpHarm_610,
  SpHarm_310]] ]: fi:
>   if d8<>0 then our_op:=[ op(our_op),
>               [d8*Convert_610^4,
>                           [SpHarm_610,SpHarm_610,SpHarm_610,
  SpHarm_610]] ]: fi:
>   our_op:
> end:


> # The following procedure Op_AM returns the SO(3) AM of an
```

```
  operator.
> # This operator is given in the form described above.
> # If the operator doesn't have definite AM then minus the largest
  value
> # is returned.
> # This procedure is only used by the procedure ACM_set_transition
> # to set glb_rat_TRopAM to be the AM of the transition operator.

> Op_AM:=proc(WOp::list(list))
>     local am,first,i,t,Wterm:

>     if nops(WOp)=0 then return 0 fi:

>     for i to nops(WOp) do
>       Wterm:=WOp[i][2]:
>       am:=0:
>       for t in Wterm do
>         if t in SpHarm_Operators then
>           am:=am+SpHarm_Table[t][3]:
>         elif t in [Xspace_Pi,Xspace_PiPi2] then
>           am:=am+2:
>         elif t = Xspace_PiPi4 then
>           am:=am+4:
>         fi:
>       od:

>       if i=1 then
>         first:=am:
>       elif first>=0 and am<>first then   #there are terms of
  different AM
>         first:=-max(first,am):
>       elif first<0 and am>-first then
>         first:=-am:
>       fi:
>     od:

>     first:
> end:


> # The following procedure Op_Parity returns the parity of an
  operator.
> # This operator is given in the form described above.
> # It returns 0 or 1 accordingly. If the operator has indeterminate
> # parity then -1 is returned.
> # This procedure is not used elsewhere.

> Op_Parity:=proc(WOp::list(list))
>     local parity,first,i,t,Wterm:

>     if nops(WOp)=0 then return 0 fi:

>     for i to nops(WOp) do
>       Wterm:=WOp[i][2]:
>       parity:=0:
>       for t in Wterm do
>         if t in [ Radial_b, Radial_bm, Radial_Db, Xspace_Pi,
>                   SpHarm_112, SpHarm_310, SpHarm_313, SpHarm_314,
  SpHarm_316,
>                   SpHarm_512, SpHarm_514, SpHarm_515, SpHarm_516,
  SpHarm_517,
>                   SpHarm_518, SpHarm_51A ] then
```

```
>           parity:=parity+1:
>         fi:
>       od:

>       if i=1 then
>         first:=irem(parity,2):
>       elif type(parity-first,odd) then    #terms of different
  parity
>         return -1
>       fi:
>     od:

>     first:
> end:

> # The following procedure Op_Tame determines whether an operator
> # doesn't contain either of
> #           Xspace_Pi, Xspace_PiPi2, Xspace_PiPi4.
> # If not, it returns true (boolean), otherwise false.
> # Later, this is used, in the case of a Hamiltonian, to indicate
  whether
> # we need only to calculate representation matrices on individual
  L-spaces.
> # (Otherwise, we need to use the full truncated Hilbert space).

> Op_Tame:=proc(WOp::list(list))
>     local parity,first,i,t,Wterm:

>     if nops(WOp)=0 then return 0 fi:

>     for i to nops(WOp) do
>       Wterm:=WOp[i][2]:
>       parity:=0:
>       for t in Wterm do
>         if t in [ Xspace_Pi, Xspace_PiPi2, Xspace_PiPi4 ] then
>           return false:
>         fi:
>       od:
>     od:

>     true:
> end:

> # The following three values specify particular (linear
  combinations of)
> # operators.
> # laplacian_op encodes the SO(5) Laplacian.
> # The latter two may be used as a check on commutation relations:
> # comm_su11_op encodes a sum of three SU(1,1) operators
> # which should produce a zero matrix;
> # comm_bdb_op encodes [d/d(beta) * beta, beta * d/d(beta)] - id.
> # which should also produce a zero matrix (note that an empty
  operator
> # product [] denotes the identity operator).


> laplacian_op:=[ [1,[Radial_D2b]],
>                       [-(2+SENIORITY*(SENIORITY+3)),[Radial_bm2]] ]:


> comm_su11_op:=[ [ 1,[Radial_Sm,Radial_Sp]],
>                       [-1,[Radial_Sp,Radial_Sm]],
```

```
>                     [-2,[Radial_SO]] ]:

> comm_bdb_op:=[ [-1,[Radial_b,Radial_Db]],
>                     [1,[Radial_Db,Radial_b]],
>                     [-1,[]] ]:


> ########################################################################
  ########
> ####-------------- Representing operators on full Xspace ----------
  ----####
> ########################################################################
  ########

> # Here, we obtain representations on the full Hilbert space by
> # combining representations on the radial and spherical spaces
> # that are obtained using the procedures given above.

> # The following procedure dimXspace returns the dimension of the
> # truncated Hilbert space for the nu range of the radial space
> # nu_min,..,nu_max, and for the spherical space, the seniority
  range
> # v_min,..,v_max, and for the angular momentum range L_min,..,
  L_max.
> # If the L_max argument is omitted, then L_max=L_min.

> dimXspace:=proc(nu_min::nonnegint,nu_max::nonnegint,
>                 v_min::nonnegint,v_max::nonnegint,
>                 L_min::nonnegint,L_max::nonnegint,$)

>     dimRadial(nu_min,nu_max)*dimSO5r3_rngVvarL(_passed[3..-1]):
> end:


> # The following procedure lbsXspace returns a list of labels [nu,v,
  alpha,L]
> # for the basis states of the truncated Hilbert space:
> # nu takes the range nu_min,..,nu_max, v takes the range v_min,..,
  v_max,
> # while L is restricted to the range L_min,..,L_max.
> # If the L_max argument is omitted, then L_max=L_min.
> # The nu label varies fastest, then alpha, then v, and L is slowest
> # (as elsewhere).
> # If the L_max argument is omitted, then L_max=L_min.

> lbsXspace:=proc(nu_min::nonnegint,nu_max::nonnegint,
>                 v_min::nonnegint,v_max::nonnegint,
>                 L_min::nonnegint,L_max::nonnegint,$)
>     local rad_labels,sph_labels;

>   rad_labels:=lbsRadial(nu_min,nu_max);             # radial labels
>   sph_labels:=lbsSO5r3_rngVvarL(_passed[3..-1]):   # SO(5) labels

>   [seq( seq( [nu,op(s)], nu in rad_labels), s in sph_labels)];
> end:


> ########################################################################
  ########


> # The procedure RepXspace below returns the (alternative SO(3)-
```

```
  reduced)
> # Matrix of polynomials of the operators listed in tables I,II and
  III,
> # together with the diagonal operators SpDiag_sqLdim and
  SpDiag_sqLdiv,
> # on the truncated subspace of the full Hilbert space specified by
> # the ranges nu_min,..,nu_max of radial states, v_min,..,v_max of
> # seniorities, and L_min,..,L_max of SO(3) angular momentum.
> # The operator is specified in the argument x_oplc, whose format is
> # the ACM encoding of operators described above (or in Section 7.3)
  .
> # The arguments anorm and lambda_base specify the parameters that
> # determine the radial basis.
> # The returned matrix elements are all floating point numbers
> # (if the list of operators is empty, the null matrix is returned).
> # The correct 4*Pi normalisation factors are included.

> # The final argument L_max is optional - if omitted then L_max=
  L_min,
> # so that a single angular momentum value is used.

> # By alternative, we mean that the matrix elements should be
  mulitplied by
> # sqrt(2*L_f+1) to get the genuine SO(3)-reduced matrix elements of
  the
> # operator in question (see (37)). These are useful in practical
  use,
> # because this 1/sqrt(2*L_f+1) appears in the Wigner-Eckart theorem
> # (see (36)). In the case of Hamiltonians, then L=M=0 and
> # (L_i M_i 0 0 | L_f M_f)=1, and the returned matrix elements give
> # the required amplitudes directly.

> # With regard to labelling [nu,v,alpha,L] of the basis vectors of
  the
> # tensor product space, the index L varies slowest (if it varies at
  all),
> # v next slowest, then alpha, with the index nu varying quickest.
> # When L varies, it does so most slowest (so that the matrices
  formed
> # for a range of L values are obtained by simply adjoining those
  obtained
> # for the individual L values).
> # This corresponds to the order of the state labels output by
  lbsXspace.

> # The values of anorm and lambda_base help to determine the radial
> # basis states (they do not affect the SO(5) action).
> # The value of lambda associated with a particular state [nu,v,
  alpha,L]
> # in the cross product space is determined by lambda_base+
  glb_lam_fun(v),
> # where the function glb_lam_fun has been previously set
> # (by ACM_set_basis_type or ACM_set_lambda_fun).
> # The initial and final bases are identical.

> RepXspace:=proc(x_oplc::list, anorm::algebraic,
  lambda_base::algebraic,
>                       nu_min::nonnegint, nu_max::nonnegint,
>                       v_min::nonnegint, v_max::nonnegint,
>                       L::nonnegint, L_max::nonnegint,$)
>       local Rmat,Pmat,i,n,Xlabels;
```

```
>   n:=nops(x_oplc);

>   Xlabels:=lbsXspace(_passed[4..-1]):    # list of all states in X-
  space.
>
>   if n=0 then    # null sum: require zero matrix
>     Rmat:=Matrix( dimXspace(_passed[4..-1]), datatype=float );
  #Null matrix
>   else

>     # first obtain rep matrix on X-space of 1st operator product

>     Rmat:=RepXspace_Prod(x_oplc[1][2],_passed[2..-1]);

>     if type(x_oplc[1][1],constant) then

>       # simply multiply by the coefficient (which is a numeric
  value)

>       MatrixScalarMultiply(Rmat,evalf(x_oplc[1][1]),inplace);
>     else

>       # post-multiply by a diagonal matrix that is formed by
  evaluating
>       # the coefficient (a function of number, seniority, alfa,
  angmom)
>       # at each state in the X_space

>       MatrixMatrixMultiply(Rmat,
>             Matrix(map(x->evalf(eval(x_oplc[1][1],
>                          [NUMBER=x[1],SENIORITY=x[2],ALFA=x[3],
  ANGMOM=x[4]])),
>                        Xlabels),
>                        shape=diagonal,scan=diagonal), inplace);
>     fi:

>     # now do similar for every other operator product - and sum
  results

>     for i from 2 to n do
>       if type(x_oplc[i][1],constant) then
>         MatrixAdd(Rmat, RepXspace_Prod(x_oplc[i][2],_passed[2..-1]
  ),
>                                        1, evalf(x_oplc[i][1]),
  inplace);
>       else
>         Pmat:=RepXspace_Prod(x_oplc[i][2],_passed[2..-1]):
>         MatrixMatrixMultiply(Pmat,
>               Matrix(map(x->evalf(eval(x_oplc[i][1],
>                          [NUMBER=x[1],SENIORITY=x[2],ALFA=x[3],
  ANGMOM=x[4]])),
>                          Xlabels),
>                          shape=diagonal,scan=diagonal),inplace);
>         MatrixAdd(Rmat,Pmat,inplace);
>       fi:
>     od:

> #    for i from 2 to n do
> #      Pmat:=RepXspace_Prod(x_oplc[i][2],_passed[2..-1]):
> #      if type(x_oplc[i][1],constant) then
> #        MatrixScalarMultiply(Pmat,evalf(x_oplc[i][1]),inplace);
> #      else
```

```
> #        MatrixMatrixMultiply(Pmat,
> #              Matrix(map(x->evalf(eval(x_oplc[i][1],
> #                           [NUMBER=x[1],SENIORITY=x[2],ALFA=x[3],
  ANGMOM=x[4]])),
> #                           Xlabels),
> #                           shape=diagonal,scan=diagonal),inplace);
> #      fi:
> #      MatrixAdd(Rmat,Pmat,inplace);
> #    od:
>   fi:

>   # now clear all the remember tables used so that the next
  calculation
>   # can start afresh (with a different Xspace).
>   # (In this list, we have given each procedure a number, and
  following that
>   #  in parentheses, the numbers of these precedures that get
  called by it:
>   #  this is useful for debugging).

>   forget(RepRadial):                # 1.
>   forget(RepRadial_param):          # 2.
>   forget(Matrix_sqrt):              # 3.
>   forget(Matrix_sqrtInv):           # 4.
>   forget(RepRadial_bS_DS):          # 5.  (1,2,3,4)
>   forget(RepRadialshfs_Prod):       # 6.  (1,5)
>   forget(RepRadial_Prod_rem):       # 7.  (6)
>   forget(RepRadial_LC_rem):         # 8.  (7)
>   forget(RepXspace_Pi):             # 9.  (8)
>   forget(RepXspace_PiPi):           # 10. (8)
>   forget(RepXspace_PiqPi):          # 11. (8)
>   forget(RepSO5_Y_rem):             # 12.
>   forget(RepSO5r3_Prod_rem):        # 13. (12)

>   Rmat;
> end:



> #
  MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
  MMMMMM

> # The procedure RepXspace_Prod below returns the (alternative SO(3)
  -reduced)
> # Matrix of a product of the operators listed in tables I,II and
  III,
> # together with the diagonal operators SpDiag_sqLdim and
  SpDiag_sqLdiv,
> # on the truncated subspace of the full Hilbert space specified by
> # the ranges nu_min,..,nu_max of radial states, v_min,..,v_max of
> # seniorities, and L_min,..,L_max of SO(3) angular momentum.
> # It is thus exactly as the procedure RepXspace above, but only
  applies
> # to products of operators (RepXspace calls this RepXspace_Prod).
> # The operator is specified in the argument x_ops, which is simply
> # a list of the symbolic names of the operators.
> # The arguments anorm and lambda_base specify the parameters that
> # determine the radial basis.
> # The returned matrix elements are all floating point numbers
> # (if the list of operators is empty, the identity matrix is
  returned).
```

```
> # The correct 4*Pi normalisation factors are included.

> # The final argument L_max is optional - if omitted then L_max=
  L_min,
> # so that a single angular momentum value is used.


> RepXspace_Prod:=proc(x_ops::list,
>                      anorm::algebraic,lambda_base::algebraic,
>                      nu_min::nonnegint,nu_max::nonnegint,
>                      v_min::nonnegint,v_max::nonnegint,
>                      L_min::nonnegint,L_max::nonnegint,$)
>     local sph_ops,nu_ops,run_Mat,xsp_Mat,this_op,up_running;
>     global Radial_Max,Spherical_Min,
>            Xspace_Pi,Xspace_PiPi2,Xspace_PiPi4,Xspace_PiqPi;

>   # Run through the list of operators left to right, storing
>   # independently those that act on the radial and spherical
  spaces.
>   # If/When we see the pi or [pi x pi] or [pi x q x pi] operators,
>   # we form the matrices and multiply them out.

>   up_running:=0:             # Flag to indicate that run_Mat
  contains summat
>   sph_ops:=[]: nu_ops:=[]:  # Accumulates operators from the left
>
>   for this_op in x_ops do

>     if member(this_op,Radial_Operators) then
>       nu_ops:=[op(nu_ops),this_op];     # store Radial Ops
>     elif member(this_op,Spherical_Operators) then
>       sph_ops:=[op(sph_ops),this_op];   # store Sph Ops
>     else

>       # we now expect an operator on the full Xspace, so we need to
  multiply
>       # out all those on the Radial and Spherical spaces so far
  accumulated.

>       if nu_ops<>[] or sph_ops<>[] then
>         xsp_Mat:=RepXspace_Twin(nu_ops,sph_ops,_passed[2..-1]):
>         nu_ops:=[]:   # used, so reset
>         sph_ops:=[]:  # ditto
>         if up_running>0 then
>           MatrixMatrixMultiply(run_Mat,xsp_Mat,inplace):
>         else          # nothing yet, so use xsp_Mat (not a copy)
>           run_Mat:=xsp_Mat:
>           up_running:=1:
>         fi:
>       fi:

>       # generate the Xspace operator as required

>       if this_op=Xspace_PiqPi then   # For operator  [pi x q x pi]_
  {v=3,L=0};
>           xsp_Mat:=RepXspace_PiqPi(_passed[2..-1]):

>       elif this_op=Xspace_PiPi2 then   # For operator  [pi x pi]_
  {v=2,L=2};
>           xsp_Mat:=RepXspace_PiPi(2,_passed[2..-1]):

>       elif this_op=Xspace_PiPi4 then   # For operator  [pi x pi]_
```

```
  {v=2,L=4};
>           xsp_Mat:=RepXspace_PiPi(4,_passed[2..-1]):

>       elif this_op=Xspace_Pi then   # For operator  [pi]_{v=1,L=2};
>           xsp_Mat:=RepXspace_Pi(_passed[2..-1]):

>       # could put other Xpsace operators here!

>       else
>         error "Operator %1 undefined", this_op:
>       fi:

>       # Now multiply in this Xspace operator

>       if up_running>0 then
>         MatrixMatrixMultiply(run_Mat,xsp_Mat,inplace);
>       else
>         run_Mat:=copy(xsp_Mat):   # need a copy because of remember
  tables
>         up_running:=1:
>       fi:
>     fi:
>   od:

>   # And we must multiply out any remaining operators.

>   if nu_ops<>[] or sph_ops<>[] then
>     xsp_Mat:=RepXspace_Twin(nu_ops,sph_ops,_passed[2..-1]):

>     if up_running>0 then
>       MatrixMatrixMultiply(run_Mat,xsp_Mat,inplace):
>     else          # nothing yet, so use xsp_Mat (not a copy)
>       run_Mat:=xsp_Mat:
>       up_running:=1:
>     fi:
>   fi:

>   if up_running=0 then   # empty operator - need identity matrix
>     run_Mat:=Matrix([seq(1,i=1..dimXspace(_passed[4..-1]))],scan=
  diagonal):
>   fi:

>   run_Mat:

> end:


> # The following procedure RepXspace_Twin does much of the work
> # for RepXspace_Prod above, and has similar arguments, except
> # that it takes two lists of operators, rad_ops and sph_ops.
> # The former is a product of the radial operators from Table I,
> # while the latter is a product of spherical operators from Table
  II
> # together with the diagonal operators SpDiag_sqLdim and
  SpDiag_sqLdiv.
> # The Matrix that is returned is the combined action on the
  truncated
> # cross-product Hilbert space that is determined by the remaining
> # arguments as above. The radial and spherical actions are
  independent,
> # although the radial states have a dependence on seniority (see
  (8)).
```

```
> # Note that the correct 4*Pi factors are applied here, so that the
> # matrix elements returned are genuine alternative SO(3)-reduced.

> # The final argument L_max is optional - if omitted then L_max=
  L_min,
> # so that a single angular momentum value is used.

> # Note that the matrix elements are all determined analytically if
> # and only if, the degree of the radial operator has the same
  parity
> # (odd or even) as the total seniority of the spherical operator.
> # Otherwise, they are determined non-analytically (through the
  taking
> # of a matrix square root).

> # It might be thought that constructing the representation matrix
> # directly from blocks coming from the tensor product, as follows,
> #  direct_Mat:=
> #  Matrix(
> #  [ seq( [ seq(
> #    `if`(sph_Mat[i2,j2]=0,Matrix(rad_dim,fill=0.0),
> #           sph_Mat[i2,j2]*evalf(RepRadial_Prod_rem(rad_ops,anorm,
> #                                lambda_base+glb_lam_fun
  (sph_labels[j2][1]),
> #                                glb_lam_fun(sph_labels[i2][1])-
> #                                           glb_lam_fun
  (sph_labels[j2][1]),
> #                                nu_min,nu_max,glb_nu_lap))),
> #                j2=1..sph_dim ) ],
> #                i2=1..sph_dim ) ],
> #         datatype=float
> #        ):
> # would be more efficient, but it is actually a lot slower (over
  10x).
> # It's even slower if we take out the 0 test.


> RepXspace_Twin:=proc(rad_ops::list, sph_ops::list,
>                      anorm::algebraic, lambda_base::algebraic,
>                      nu_min::nonnegint, nu_max::nonnegint,
>                      v_min::nonnegint, v_max::nonnegint,
>                      L_min::nonnegint, L_max::nonnegint,$)
>     local j2,i2,j1,i1,jdisp,idisp,lambda_disp_init,lambda_disp_fin,
>           rad_dim,rad_Mat,st,
>           sph_dim,sph_labels,sph_Mat,sph_ME,
>           direct_Mat;
>     global glb_lam_fun,glb_nu_lap:=0,glb_time:


>   # Obtain dim, labels, and representation matrix for the spherical
  operator

>   sph_dim:=dimSO5r3_rngVvarL(_passed[7..-1]);
>   sph_labels:=lbsSO5r3_rngVvarL(_passed[7..-1]);
>   sph_Mat:=RepSO5r3_Prod_rem(sph_ops,_passed[7..-1]);

>   # Now include the (4*Pi) factors in the latter:

>   sph_Mat:=MatrixScalarMultiply(sph_Mat,
>               evalf(Convert_red^NumSO5r3_Prod(sph_ops)));

>   # dimension of radial space:
```

```
>   rad_dim:=dimRadial(nu_min,nu_max);

>   # Now form the direct product representations on the space of
>   # dimension sph_dim*rad_dim.

>   direct_Mat:=Matrix(sph_dim*rad_dim,datatype=float);

>   # Place the entries one-by-one into the direct product matrix.
>   # (j1;j2) is initial state, (i1;i2) is final state
>   # 1st label is radial and varies quickest, 2nd is spherical and
  slowest.

>   # For the radial (nu) part, the rep matrix depends on the initial
>   # and final values of v: these determine the lambdas mapped
  between.

> #  st:=time():

>   for j2 to sph_dim do
>     jdisp:=(j2-1)*rad_dim:
>     lambda_disp_init:=glb_lam_fun(sph_labels[j2][1]):

>   for i2 to sph_dim do
>     idisp:=(i2-1)*rad_dim:
>     lambda_disp_fin:=glb_lam_fun(sph_labels[i2][1]):
>     sph_ME:=sph_Mat[i2,j2]:

>     if sph_ME=0 then  # skip zero cases of spherical MEs.
>       next
>     fi:

>     # Form representation on the radial space, taking account
>     # of the correct lambda variation. Use the version with the
>     # remember option because it might need to be reused here.

>     rad_Mat:=RepRadial_Prod_rem(rad_ops,anorm,
>                                 lambda_base+lambda_disp_init,
>                                 lambda_disp_fin-lambda_disp_init,
>                                 nu_min,nu_max,glb_nu_lap):

>     for i1 to rad_dim do
>     for j1 to rad_dim do
>       direct_Mat[idisp+i1,jdisp+j1]:=evalf(rad_Mat[i1,j1]*sph_ME);
>     od: od:
>   od: od:

>   direct_Mat;
> end:


> # The following procedure RepXpsace_Pi returns the Matrix
  representation
> # of the pi/(-i\hbar) operator on the truncated Hilbert space
> # determined by the arguments as above.
> # The returned matrix elements are alternative SO(3)-reduced matrix
  elements.
> # They are calculated using (53). The result should be
  antihermitian.

> # If an exotic coefficient (such as a function of NUMBER,
  SENIORITY,
> # ANGMOM or ALFA) is required, the procedure RepXspace can be used
```

```
> # (it calls this one).

> RepXspace_Pi:=proc( anorm::algebraic, lambda_base::algebraic,
>                     nu_min::nonnegint, nu_max::nonnegint,
>                     v_min::nonnegint, v_max::nonnegint,
>                     L_min::nonnegint, L_max::nonnegint,$)
>     option remember;
>     local j2,i2,j1,i1,jdisp,idisp,lambda_disp_init,lambda_disp_fin,
>           v_init,al_init,L_init,v_fin,al_fin,L_fin,v_chg,
>           sph_dim,sph_labels,
>           rad_dim,rad_Mat,direct_Mat,CG2;
>     global glb_lam_fun,Radial_Db,Radial_bm;

>   # dimension of radial space:

>   rad_dim:=dimRadial(nu_min,nu_max);

>   # Obtain dim and labels for S5 space.

>   sph_dim:=dimSO5r3_rngVvarL(_passed[5..-1]);
>   sph_labels:=lbsSO5r3_rngVvarL(_passed[5..-1]);

>   # Will form the representation on the sph_dim*rat_dim dimensional
>   # direct product space in the following Matrix, which is
  returned.

>   direct_Mat:=Matrix(sph_dim*rad_dim,datatype=float);

>   # Place the entries one-by-one into the direct product matrix.
>   # (j1;j2) is initial state, (i1;i2) is final state.
>   # 1st label is radial, 2nd is spherical, and varies slowest.

>   # For the radial (nu) part, the rep matrix depends on the initial
>   # and final values of v: these determine the lambdas mapped
  between.

>   for j2 to sph_dim do
>     v_init:=sph_labels[j2][1]:   # seniority of initial state
>     al_init:=sph_labels[j2][2]:  # alpha of same
>     L_init:=sph_labels[j2][3]:   # L of same
>     jdisp:=(j2-1)*rad_dim:
>     lambda_disp_init:=glb_lam_fun(v_init):

>   for i2 to sph_dim do
>     L_fin:=sph_labels[i2][3]:    # L of final state
>     if L_fin-L_init>2 or L_fin-L_init<-2 then next fi:  # zero
  becos q has L=2.

>     v_fin:=sph_labels[i2][1]:    # seniority of final state
>     al_fin:=sph_labels[i2][2]:   # alpha of same
>     v_chg:=v_fin-v_init:         # change in seniority
>     idisp:=(i2-1)*rad_dim:
>     lambda_disp_fin:=glb_lam_fun(v_fin):


>     # Now obtain the (SO(5) reduced) representation matrix between
  these
>     # subspaces having constant spherical labels by treating
  separately
>     # the cases v_chg = 1 (using (53a)) and -1 (using (53b)).

>     if v_chg = 1 then
```

```
>        rad_Mat:=RepRadial_LC_rem(
>                        [ [1,[Radial_Db]],[-v_init-2,[Radial_bm]] ],
>                        anorm, lambda_base+lambda_disp_init,
>                        lambda_disp_fin-lambda_disp_init, nu_min,
  nu_max):

>       # multiply this rad_Mat radial action by the SO(5) reduced ME
>       # from (45).
>       # Note that we can't do this 'inplace' because this will
  affect the
>       # remember tables in RepRadial_LC_rem.

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,evalf(Qred_p1(v_init)))
  ;

>     elif v_chg = -1 then

>        rad_Mat:=RepRadial_LC_rem(
>                        [ [1,[Radial_Db]],[v_init+1,[Radial_bm]] ],
>                        anorm, lambda_base+lambda_disp_init,
>                        lambda_disp_fin-lambda_disp_init, nu_min,
  nu_max):

>       # multiply this nu_Mat radial action by the SO(5) reduced ME

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,evalf(Qred_m1(v_init)))
  ;

>     else
>       next           # skip this (j1,i1) case (because it is zero).
>     fi:

>     # The "alternative" SO(3)-reduced matrix element is obtained
  from the
>     # SO(5)-reduced ME calculated above by multiplying with the
  following
>     # (we could also multiply by sqrt(dimSO3(L_fin)) here to get
  genuine
>     #  SO(3)-reduced matrix elements):

>     CG2:=CG_SO5r3(v_init,al_init,L_init,1,1,2,v_fin,al_fin,L_fin):

>     # fill in the Xspace elements for these constant spherical
>     # parameters (i2,j2).

>     for i1 to rad_dim do
>     for j1 to rad_dim do
>       direct_Mat[idisp+i1,jdisp+j1]:=evalf(CG2*rad_Mat[i1,j1]);
>     od: od:
>   od: od:

>   direct_Mat:
> end:


> # The following procedure RepXpsace_PiPi returns Matrix
  representations
> # of the operators
> #            [pi x pi]_(v=2,L=2)          [pi x pi]_(v=2,L=4)
> #            -------------------   and    ------------------- ,
> #                  hbar^2                       hbar^2
> # for PiPi_L = 2 or 4 respectively, on the truncated Hilbert spaces
```

```
> # determined by the other arguments as above.
> # The returned matrix elements are alternative SO(3)-reduced matrix
  elements.
> # They are calculated using (D11), with (D3).
> # The result deviates slightly from being hermitian due to
  truncation effects.

> # If an exotic coefficient (such as a function of NUMBER,
  SENIORITY,
> # ANGMOM or ALFA) is required, the procedure RepXspace can be used
> # (it calls this one).

> RepXspace_PiPi:=proc(PiPi_L::nonnegint,
>                      anorm::algebraic, lambda_base::algebraic,
>                      nu_min::nonnegint, nu_max::nonnegint,
>                      v_min::nonnegint, v_max::nonnegint,
>                      L_min::nonnegint, L_max::nonnegint,$)
>     option remember;
>     local j2,i2,j1,i1,jdisp,idisp,lambda_disp_init,lambda_disp_fin,
>           v_init,al_init,L_init,v_fin,al_fin,L_fin,v_chg,
>           sph_dim,sph_labels,
>           rad_dim,rad_Mat,direct_Mat,CG2;
>     global glb_lam_fun,Radial_D2b,Radial_bm2,Radial_bDb;

>   # dimension of radial space:

>   rad_dim:=dimRadial(nu_min,nu_max);

>   # Obtain dim and labels for S5 space.

>   sph_dim:=dimSO5r3_rngVvarL(_passed[6..-1]);
>   sph_labels:=lbsSO5r3_rngVvarL(_passed[6..-1]);

>   # Will form the representation on the sph_dim*rat_dim dimensional
>   # direct product space in the following Matrix, which is
  returned.

>   direct_Mat:=Matrix(sph_dim*rad_dim,datatype=float);

>   # Place the entries one-by-one into the direct product matrix.
>   # (j1;j2) is initial state, (i1;i2) is final state.
>   # 1st label is radial, 2nd is spherical, and varies slowest.

>   # For the radial (nu) part, the rep matrix depends on the initial
>   # and final values of v: these determine the lambdas mapped
  between.

>   for j2 to sph_dim do
>     v_init:=sph_labels[j2][1]:   # seniority of initial state
>     al_init:=sph_labels[j2][2]:  # alpha of same
>     L_init:=sph_labels[j2][3]:   # L of same
>     jdisp:=(j2-1)*rad_dim:
>     lambda_disp_init:=glb_lam_fun(v_init):

>   for i2 to sph_dim do
>     L_fin:=sph_labels[i2][3]:    # L of final state
>     if L_fin-L_init>PiPi_L or L_fin-L_init<-PiPi_L then next fi: #
  zero

>     v_fin:=sph_labels[i2][1]:    # seniority of final state
>     al_fin:=sph_labels[i2][2]:   # alpha of same
>     v_chg:=v_fin-v_init:         # change in seniority
```

```
>     idisp:=(i2-1)*rad_dim:
>     lambda_disp_fin:=glb_lam_fun(v_fin):


>     # Now obtain the (SO(5) reduced) representation matrix on this
>     # subspace with constant spherical labels by treating
  separately
>     # the cases v_chg = 2 and -2 and 0 (all using (D11) ).

>     if v_chg = 2 then

>        rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_D2b]],
>                                     [(v_init+2)*(v_init+4),
  [Radial_bm2]],
>                                     [-2*v_init-5,[Radial_bm2,
  Radial_bDb]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):

>       # multiply this nu_Mat radial action by the SO(5) reduced ME
>       # from (D2) (the minus sign comes from the i^2 (*hbar^2) )
>       # Note that we can't do this 'inplace' because this will
  affect the
>       # remember tables in RepRadial_LC_rem.

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,-evalf(QxQred_p2
  (v_init)));

>     elif v_chg = -2 then

>        rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_D2b]],
>                                     [(v_init-1)*(v_init+1),
  [Radial_bm2]],
>                                     [2*v_init+1,[Radial_bm2,
  Radial_bDb]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):

>       # multiply this nu_Mat radial action by the SO(5) reduced ME

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,-evalf(QxQred_m2
  (v_init)));

>     elif v_chg = 0 then

>        rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_D2b]],
>                                     [-(v_init+1)*(v_init+2),
  [Radial_bm2]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):

>       # multiply this nu_Mat radial action by the SO(5) reduced ME

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,-evalf(QxQred_0(v_init)
  ));
>     else
>       next           # skip this (j1,i1) case (because it is zero).
>     fi:
```

```
>     # The "alternative" SO(3)-reduced matrix element is obtained
  from the
>     # SO(5)-reduced ME calculated above by multiplying with the
  following
>     # (we could multiply by sqrt(dimSO3(L_fin)) here to get genuine
>     #  SO(3)-reduced matrix elements)
>     # (PiPi_L should be 2 or 4 here, else we'll get 0):

>     CG2:=CG_SO5r3(v_init,al_init,L_init,2,1,PiPi_L,v_fin,al_fin,
  L_fin):

>     # fill in the Xspace elements for these constant spherical
>     # parameters (i2,j2).

>     for i1 to rad_dim do
>     for j1 to rad_dim do
>       direct_Mat[idisp+i1,jdisp+j1]:=evalf(CG2*rad_Mat[i1,j1]);
>     od: od:
>   od: od:

>   # This now correct for PiPi_L=4, but the sign needs to change for
  PiPi_L=2
>   # (see sign in (D3)).

>   if PiPi_L=2 then
>       MatrixScalarMultiply(direct_Mat,-1,inplace);
>   fi:

>   direct_Mat:
> end:


> # The following procedure RepXpsace_Pi returns the Matrix
  representation
> # of the operator
> #                           [pi x q x pi]_(v=3,L=0)
> #                           ----------------------
> #                                   hbar^2
> # on the truncated Hilbert space determined by the arguments as
  above.
> # The returned matrix elements are alternative SO(3)-reduced matrix
  elements.
> # They are calculated using (D12) & (D14).
> # The result deviates slightly from being hermitian due to
  truncation effects.

> # If an exotic coefficient (such as a function of NUMBER,
  SENIORITY,
> # ANGMOM or ALFA) is required, the procedure RepXspace can be used
> # (it calls this one).


> RepXspace_PiqPi:=proc( anorm::algebraic, lambda_base::algebraic,
>                        nu_min::nonnegint, nu_max::nonnegint,
>                        v_min::nonnegint, v_max::nonnegint,
>                        L_min::nonnegint, L_max::nonnegint,$)
>     option remember;
>     local j2,i2,j1,i1,jdisp,idisp,lambda_disp_init,lambda_disp_fin,
>           v_init,al_init,L_init,v_fin,al_fin,L_fin,v_chg,
>           sph_dim,sph_labels,
>           rad_dim,rad_Mat,rad_Mat2,direct_Mat,CG2;
>     global glb_lam_fun,Radial_bm2,Radial_D2b,Radial_bDb,
```

```
>                        Radial_b,Radial_Db,Radial_bm;

>   # dimension of radial space:

>   rad_dim:=dimRadial(nu_min,nu_max);

>   # Obtain dim and labels for S5 space.

>   sph_dim:=dimSO5r3_rngVvarL(_passed[5..-1]);
>   sph_labels:=lbsSO5r3_rngVvarL(_passed[5..-1]);

>   # Will form the representation on the sph_dim*rat_dim dimensional
>   # direct product space in the following Matrix, which is
  returned.

>   direct_Mat:=Matrix(sph_dim*rad_dim,datatype=float);

>   # Place the entries one-by-one into the direct product matrix.
>   # (j1;j2) is initial state, (i1;i2) is final state.
>   # 1st label is radial, 2nd is spherical, and varies slowest.

>   # For the radial (nu) part, the rep matrix depends on the initial
>   # and final values of v: these determine the lambdas mapped
  between.

>   for j2 to sph_dim do
>     v_init:=sph_labels[j2][1]:   # seniority of initial state
>     al_init:=sph_labels[j2][2]:  # alpha of same
>     L_init:=sph_labels[j2][3]:   # L of same
>     jdisp:=(j2-1)*rad_dim:
>     lambda_disp_init:=glb_lam_fun(v_init):

>   for i2 to sph_dim do
>     L_fin:=sph_labels[i2][3]:    # L of final state
>     if L_fin<>L_init then next fi:   # need L's equal for this
  operator
>     v_fin:=sph_labels[i2][1]:    # seniority of final state
>     al_fin:=sph_labels[i2][2]:   # alpha of same
>     v_chg:=v_fin-v_init:         # change in seniority
>     idisp:=(i2-1)*rad_dim:
>     lambda_disp_fin:=glb_lam_fun(v_fin):

>     # Now obtain the (SO(5) reduced) representation matrix on this
>     # subspace with constant spherical labels by treating
  separately
>     # the cases v_chg = 3 (using (D12a)) and -3 (using (D12b))
>     # and 1 (using (D14a)) and -1 (using (D14b)).
>     # The SO(5)>SO(3) CG coefficient is left until the end.
>     # (Note that here, we use a sixth parameter 1 to
  RepRadial_LC_rem:
>     # this temporarily expands the size of the radial space used -
>     # to get a more accurate matrix).

>     if v_chg = 3 then

>        rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_b,Radial_D2b]],
>                                     [(v_init+2)*(v_init+4),
  [Radial_bm]],
>                                     [-2*v_init-5,[Radial_Db]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max,1):
```

```
>       # multiply this rad_Mat radial action by the SO(5) reduced ME
>       # from (D4) (the minus sign in (D4) cancels that in (D12),
  which
>       # comes from the i^2 (*hbar^2) ).
>       # Note that we can't do this 'inplace' because this will
  affect the
>       # remember tables in RepRadial_LC_rem.

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,evalf(QxQxQred_p3
  (v_init)));

>     elif v_chg = -3 then

>         rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_b,Radial_D2b]],
>                                      [(v_init-1)*(v_init+1),
  [Radial_bm]],
>                                       [2*v_init+1,[Radial_Db]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max,1):

>       # multiply this rad_Mat radial action by the SO(5) reduced ME

>       rad_Mat:=MatrixScalarMultiply(rad_Mat,evalf(QxQxQred_m3
  (v_init)));

>     elif v_chg = 1 then

>         rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_b,Radial_D2b]],
>                                      [-(v_init+1)*(v_init+2),
  [Radial_bm]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):
>         rad_Mat2:=RepRadial_LC_rem( [ [1,[Radial_Db]],
>                                       [-v_init-2,[Radial_bm]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):

>       # combine rad_Mat and rad_Mat2 radials with appropriate SO(5)
  reduced MEs
>       # Note that for the first term, the minus sign in (D4)
  cancels one in (D14a).

>       rad_Mat:=MatrixAdd(rad_Mat,rad_Mat2,
>                  +evalf(QxQxQred_p1(v_init)),
>                  +evalf((2*v_init+5)*QixQxQred(v_init,v_fin,
  v_fin+1)))

>     elif v_chg = -1 then

>         rad_Mat:=RepRadial_LC_rem( [ [1,[Radial_b,Radial_D2b]],
>                                      [-(v_init+1)*(v_init+2),
  [Radial_bm]] ],
>                                  anorm, lambda_base+lambda_disp_init,
>                                  lambda_disp_fin-lambda_disp_init,
>                                  nu_min, nu_max):
>         rad_Mat2:=RepRadial_LC_rem( [ [1,[Radial_Db]],
>                                       [v_init+1,[Radial_bm]] ],
>                                  anorm, lambda_base+lambda_disp_init,
```

```
>                                   lambda_disp_fin-lambda_disp_init,
>                                   nu_min, nu_max):

>       # combine rad_Mat and rad_Mat2 radials with appropriate SO(5)
  reduced MEs

>       rad_Mat:=MatrixAdd(rad_Mat,rad_Mat2,
>                  +evalf(QxQxQred_m1(v_init)),
>                  -evalf((2*v_init+1)*QixQxQred(v_init,v_fin,
  v_fin-1)))

>     else
>       next           # skip this (j1,i1) case (because it is zero).
>     fi:

>     # The "alternative" SO(3)-reduced matrix element is obtained
  from the
>     # SO(5)-reduced ME calculated above by multiplying with the
  following
>     # (we could multiply by sqrt(dimSO3(L_fin)) here to get genuine
>     #  SO(3)-reduced matrix elements)

>     CG2:=CG_SO5r3(v_init,al_init,L_init,3,1,0,v_fin,al_fin,L_fin):

>     # fill up the Xspace elements for these constant spherical
>     # parameters (i2,j2).

>     for i1 to rad_dim do
>     for j1 to rad_dim do
>       direct_Mat[idisp+i1,jdisp+j1]:=evalf(CG2*rad_Mat[i1,j1]);
>     od: od:
>   od: od:

>   direct_Mat:
> end:


> # The following procedure QixQxQred returns the genuine SO(5)
  reduced
> # matrix elements
> #          <v_f ||| [[Q^+ x Q x Q]]^3 ||| v_i>,              (**)
> #          <v_f ||| [[Q^- x Q x Q]]^3 ||| v_i>,              (***)
> # for v_f=v_i +/- 1, calculated by making use of (D15) & (D17)
> # (Note that (D15) is independent of L and alpha_i and alpha_f,
> #  which (D17) exploits by choosing particular values of these.)
> # Because v_f=v_i +/- 1, there are actually four cases,
> # where, for (**), the "intermediate" seniority v_int=v_f-1,
> # and for (***), v_int=v_f+1.
> # However, only two cases are required in the ACM. These are
> # those considered in (D17), which are for which v_f-v_i=v_int-v_f.
> # The return value is a mixture of surds and floats, and should
> # be acted upon by evalf to give a sensible value.

> # Summing over the two possible v_int should give the SO(5) reduced
> # matrix element - <v_f||| [QxQxQ]^3 |||v_i>  ( [QxQxQ]^3 prop to
  Y^3_610 )

> QixQxQred:=proc(v_i::nonnegint,v_f::nonnegint,v_int::nonnegint)
>   local mediates,L_i;

>   L_i:=2*min(v_i,v_f):  # initial and final value of L
```

```
>   # obtain the list of intermediate states with seniority v_int

>   mediates:=lbsSO5r3_rngL(v_int,L_i-2,L_i+2):

>   # Using (D16), sum over them to get a SO(3) reduced matrix
  element.
>   # Note that Q=4*Pi/sqrt(15) * Y112; [QxQ]_2=-4*Pi*sqrt(2/105) *
  Y212.
>   # (note that a factor of sqrt(dimSO3(L_i)) is cancelled with
  below.)

>   - ME_SO5red(v_f,1,v_int) * ME_SO5red(v_int,2,v_i) * sqrt(2/7) /
  15
>     * add( CG_SO5r3(v_int,m[2],m[3],1,1,2,v_f,1,L_i)
>              * CG_SO5r3(v_i,1,L_i,2,1,2,v_int,m[2],m[3])
>              * sqrt(dimSO3(m[3]))  * (-1)^m[3], m in mediates)

>   # then convert this to a SO(5) reduced matrix element by dividing

>     /CG_SO5r3(v_i,1,L_i,3,1,0,v_f,1,L_i)/sqrt(5*dimSO3(L_i)):

> end;


> # The four cases are also implemented separately by the following
> # procedures (we don't use these procedures, but they are
  instructive!)
> # The correspondence is
> #    QpxQxQred_p1(v)  <->  QixQxQred(v,v+1,v)
> #    QmxQxQred_p1(v)  <->  QixQxQred(v,v+1,v+2)
> #    QpxQxQred_m1(v)  <->  QixQxQred(v,v-1,v-2)
> #    QmxQxQred_m1(v)  <->  QixQxQred(v,v-1,v)

> # Note that, although all v>=0 are accepted as arguments for each,
> # the first two don't make physical sense for v=0,
> # and the last two don't make physical sense for v<=1 (both give
  errors).
> # However, these exceptional values aren't required.

> QpxQxQred_p1:=proc(v::nonnegint)
>   local mediates,L1;

>   L1:=2*v:  # initial value of L

>   # obtain the list of intermediate states

>   mediates:=lbsSO5r3_rngL(v,L1-2,L1):

>   # sum over them to get a singly reduced matrix element

>   Qred_p1(v) * QxQred_0(v) *
>     add( CG_SO5r3(v,m[2],m[3],1,1,2,v+1,1,L1)
>            * CG_SO5r3(v,1,L1,2,1,2,v,m[2],m[3])
>            * sqrt(dimSO3(m[3]))  * (-1)^m[3], m in mediates)

>   # then convert this to a doubly reduced matrix element by
  dividing

>     /CG_SO5r3(v,1,L1,3,1,0,v+1,1,L1)/sqrt(5*dimSO3(L1)):

> end;
```

```
> QmxQxQred_p1:=proc(v::nonnegint)
>   local mediates,L1;

>   L1:=2*v:  # initial value of L

>   # obtain the list of intermediate states

>   mediates:=lbsSO5r3_rngL(v+2,L1-2,L1+2):

>   # sum over them to get a singly reduced matrix element

>   Qred_m1(v+2) * QxQred_p2(v) *
>     add( CG_SO5r3(v+2,m[2],m[3],1,1,2,v+1,1,L1)
>           * CG_SO5r3(v,1,L1,2,1,2,v+2,m[2],m[3])
>           * sqrt(dimSO3(m[3])) * (-1)^m[3] , m in mediates)

>   # then convert this to a doubly reduced matrix element by
  dividing

>     /CG_SO5r3(v,1,L1,3,1,0,v+1,1,L1)/sqrt(5*dimSO3(L1)):

> end;

> QpxQxQred_m1:=proc(v::posint)
>   local mediates,L1;

>   L1:=2*v-2:  # initial value of L

>   # this case has only one intermediate state [v-1,1,2v-4]

>   # get singly reduced matrix element

>   Qred_p1(v-2) * QxQred_m2(v)
>           * CG_SO5r3(v-2,1,L1-2,1,1,2,v-1,1,L1)
>           * CG_SO5r3(v,1,L1,2,1,2,v-2,1,L1-2)
>           * sqrt(dimSO3(L1-2))

>   # then convert this to a doubly reduced matrix element by
  dividing

>     /CG_SO5r3(v,1,L1,3,1,0,v-1,1,L1)/sqrt(5*dimSO3(L1)):

> end;

> QmxQxQred_m1:=proc(v::posint)
>   local mediates,L1;

>   L1:=2*v-2:  # initial value of L

>   # obtain the list of intermediate states

>   mediates:=lbsSO5r3_rngL(v,L1-2,L1+2):

>   # sum over them to get a singly reduced matrix element

>   Qred_m1(v) * QxQred_0(v) *
>     add( CG_SO5r3(v,m[2],m[3],1,1,2,v-1,1,L1)
>           * CG_SO5r3(v,1,L1,2,1,2,v,m[2],m[3])
>           * sqrt(dimSO3(m[3])) * (-1)^m[3] , m in mediates)

>   # then convert this to a doubly reduced matrix element by
  dividing
```

```
>     /CG_SO5r3(v,1,L1,3,1,0,v-1,1,L1)/sqrt(5*dimSO3(L1)):

> end;


> ##################################################################
  ########
> ####---------- Diagonalisation and Eigenbasis transformation ------
  ----####
> ##################################################################
  ########

> # The following procedure DigXspace represents the operator encoded
> # in ham_op on the truncated Hilbert space specified by the other
  arguments, and then diagonalises it.
> # The return value is a quartet of values
> #           [eigen_vals, eigen_bases, Xparams, Lvals].
> # Here Xparams lists the parameters [anorm,lambda,nu_min,nu_max,
  v_min,v_max]
> # (without L). Here, Lvals is a list of the values of angular
  momentum L
> # between L_min..L_max which are of non-zero dimension in the
  truncated
> # Hilbert space. The elements of the other two values pertain to
  these
> # values of L. eigen_vals is a list, each element of which contains
  a list
> # of eigenvalues in a constant L-space.
> # eigen_bases is the list of transformation matrices to the
  eigenspaces.
> # It's probably not a good idea to display this output!
> # The values in eigen_vals are probably best displayed using
> # the Show_Eigs procedure below.
> # The other output values may be directly used as input to the
> # procedure AmpXspeig to calculate transition rates/amplitudes.

> # Note that diagonalisation (via the procedure Eigenfiddle below)
> # is always carried out separately on each L-space.
> # However, the calculation of the representation itself might not
  be done
> # separately on these spaces if the Hamiltonian encoded by ham_op
> # makes use of the contraction features.

> DigXspace:=proc(ham_op::list,
>                       anorm::algebraic, lambda_base::algebraic,
>                       nu_min::nonnegint, nu_max::nonnegint,
>                       v_min::nonnegint, v_max::nonnegint,
>                       L_min::nonnegint, L_max::nonnegint,$)
>       local LL,LLM,eigen_bases,eigen_result,Lvals,
>             Xparams,eigen_vals,rep_matrix,L_matrix,
>             rad_dim,sph_dim,Lstart;

>   if _npassed<9 then  # no L_max
>     LLM:=L_min;
>   else
>     LLM:=L_max
>   fi:

>   Xparams:=[anorm,lambda_base,nu_min,nu_max,v_min,v_max];  # 6
  params (no L)

>   # Use Lvals to store those L with non-zero dimension.
```

```
>   # Then only diagonalise on these spaces.

>   Lvals:=[]:
>   eigen_vals:=[];       # for eigenvalues for these L
>   eigen_bases:=[];      # corresponding matrices of column vectors

>   # We decide now whether to calculate the representation matrix on
  the
>   # whole (truncated) space, or individually on the component L-
  spaces.
>   # The latter is sufficient if the Hamiltonian (AM=0) contains no
>   # terms of non-zero angular momentum.
>   # However, in both cases, we still diagonalise only the
  individual
>   # L-spaces.

>   if Op_Tame(ham_op) then  # work on L-spaces seperately...

>     for LL from L_min to LLM do
>       sph_dim:=dimSO5r3_rngV(v_min,v_max,LL):
>       if sph_dim>0 then
>         Lvals:=[op(Lvals),LL];

>         L_matrix:=RepXspace(ham_op,op(Xparams),LL);
>         eigen_result:=Eigenfiddle(L_matrix);

>         # store eigenvalue lists, one LL at a time.

>         eigen_vals:=[op(eigen_vals),eigen_result[1]];

>         # store matrix of eigenvectors (inverses to be obtained
  elsewhere)

>         eigen_bases:=[op(eigen_bases),eigen_result[2]];
>       fi:
>     od:

>   else
>     rep_matrix:=RepXspace(ham_op,op(Xparams),L_min,LLM);
>     rad_dim:=dimRadial(nu_min,nu_max);
>     Lstart:=1:

>     for LL from L_min to LLM do
>       sph_dim:=dimSO5r3_rngV(v_min,v_max,LL):
>       if sph_dim>0 then
>         Lvals:=[op(Lvals),LL];

>         L_matrix:=SubMatrix(rep_matrix,[Lstart..Lstart+rad_dim*
  sph_dim-1],
>                                        [Lstart..Lstart+rad_dim*
  sph_dim-1]):
>         eigen_result:=Eigenfiddle(L_matrix);

>         eigen_vals:=[op(eigen_vals),eigen_result[1]];

>         eigen_bases:=[op(eigen_bases),eigen_result[2]];
>         Lstart:=Lstart+rad_dim*sph_dim;
>       fi:
>     od:
>   fi:

>   [eigen_vals, eigen_bases, Xparams, Lvals];
```

```
> end;


> # The following procedure Eigenfiddle diagonalises the Matrix which
  is
> # passed to it, and returns a pair
> #                           [eigs_list,basis_Mat].
> # The first element of this pair is a list of the eigenvalues in
> # ascending order, and the second element of the pair is the matrix
  P
> # which transforms the original matrix H to the diagonal matrix P^
  {-1}HP
> # whose diagonal elements are those given in the first element of
  the pair.
> # Thus, its columns are the eigenvectors corresponding to those
  eigenvalues.

> # The matrix (H) that is passed to Eigenfiddle is diagonalised
  using
> # Maple's Eigenvectors procedure.
> # This matrix ought to be Hermitian but might not be because of
  truncation
> # effects. Thus, before being diagonalised, it is averaged with its
  transpose
> # to ensure that it is symmetric, and therefore yields real
  eigenvalues.

> # Note that Maple attempts to diagonalise retaining the datatype
> # of the passed Matrix. If this datatype is not a float, then
> # the procedure is very slow. For ACM calculations, we should
> # therefore ensure that Hmatrix has float entries.


> Eigenfiddle:=proc(Hmatrix::Matrix,$)
>     local i,n,real_eigens,eigenstuff,eigen_order;
>
>   n:=RowDimension(Hmatrix);

>   # The Maple function Eigenvectors returns a pair, the first of
>   # which is a list of eigenvalues, and the second is a matrix
>   # whose columns are the corresponding eigenvectors.
>   # We ensure that the Matrix being processed is diagonal by
>   # averaging it with its transpose.

>   eigenstuff:=Eigenvectors(Matrix(n,n,(i,j)->(Hmatrix[i,j]+Hmatrix
  [j,i])/2,
>                                    scan=diagonal[upper],shape=
  symmetric));

>   # The following list contains pairs [eig,i], where i is the index
>   # in the list. The idea is to sort the eigenvalues into
  increasing
>   # order, but keep track of their original i's, so that we can
>   # then use the same order for the eigenvectors.
>
>   real_eigens:=[seq([eigenstuff[1][i],i],i=1..n)];

>   # Now sort these into increasing values of the eigenvalues using
>   # the pair_order function defined below.

>   real_eigens:=sort(real_eigens,pair_order);
```

```
>   # Get the index order - to be applied to the eigenvectors below.

>   eigen_order:=map2(op,2,real_eigens);

>   # Return pair,
>   #   element 1 lists all (real) e-values
>   #   element 2 is a transformation matrix, with the
>   #              columns e-vectors of above e-values.

>   [ map2(op,1,real_eigens), Matrix([Column(eigenstuff[2],
  eigen_order)]) ];
> end:


> # The following procedure pair_order compares two lists, by
  comparing
> # their first elements: it returns true if the first element of the
> # first argument is less than the first element of the second.
> # This is only used by Eigenfiddle above, in order to order
  eigenvectors.

> pair_order:=proc(eigenpair1::list(numeric),eigenpair2::list
  (numeric))
>   evalb(eigenpair1[1]<eigenpair2[1]);
> end:


> # The following procedure AmpXspeig represents the operator encoded
> # in tran_op on the truncated Hilbert space specified by the
  elements
> # of Xparams and Lvals, and then transforms it to the basis
  specified
> # in eigen_bases (which is possibly an eigenbasis of some other
  operator).
> # The return value is a "block matrix" of Matrices, each of which
> # gives the alternative SO(3)-reduced transition amplitudes
> #
> #           <n_f,L_f || W || n_i,L_i>
> #           -------------------------
> #                sqrt (2*L_f+1)
> #
> # between the set of states of angular momentum L_i and the set of
  states
> # of angular momentum L_f (the states are indexed by n_i & n_f
  respectively).
> # Note that the (i1,i2) block matrix element corresponds to the
> # angular momenta Lvals[i1] and Lvals[i2].

> # The output from this procedure is probably best displayed using
  the
> # Show_Rats and Show_Amps procedures below.
> # These apply the correct functions to the raw matrix elements,
> # and also apply current values of the scaling factors.

> AmpXspeig:=proc(tran_op::list, eigen_bases::list,
>                                          Xparams::list,
  Lvals::list)
>       local i,j,LL,L_min,L_max,L_dims,L_ends,L_count,
>             eigen_invs,tran_mat,block_tran_mat;

>   L_count:=nops(Lvals);
```

```
>   if Lcount=0 then return fi:   # nothing to do

>   L_min:=Lvals[1]:
>   L_max:=Lvals[L_count]:

>   # For the alternative SO(3)-reduced transition operator, form the
>   # transformation matrix encompassing all L values
>   # (we cut it into blocks below).

>   tran_mat:=RepXspace(tran_op,op(Xparams),L_min,L_max);

>   # Obtain the sizes of the blocks (one for each good L value).

>   L_dims:=[seq(dimXspace(op(3..6,Xparams),LL),LL in Lvals)];
>   L_ends:=[seq(dimXspace(op(3..6,Xparams),L_min,LL),LL in Lvals)];

>   # Form the block matrix, each element of which is itself a
  matrix.
>   # The (i,j) block is of size L_dims[i] x L_dims[j].

>   block_tran_mat:=Matrix(L_count,
>         (i,j)->SubMatrix(tran_mat,[L_ends[i]-L_dims[i]+1..L_ends[i]
  ],
>                                    [L_ends[j]-L_dims[j]+1..L_ends[j]
  ]) );

>   # Here we simply transform to the basis (an eigenbasis) specified
  in the
>   # matrices eigen_basis, after first forming inverse transition
  matrices.
>   # (calculating the inverse matrices each time this procedure is
>   #  called is probably not inefficient because this procedure will
>   #  usually only be called once for a given set of parameters.)

>   eigen_invs:=map(MatrixInverse,eigen_bases);

>   Matrix(L_count, (i,j)->eigen_invs[i].block_tran_mat[i,j].
  eigen_bases[j]);

> end;


> ##################################################################
  ########

> # The following procedure Show_Eigs displays in a convenient format
> # lists of eigenvalues. It is designed to use directly the
> # items "eigenvals" and "Lvals" of the lists returned by the
> # procedures DigXspace, ACM_Scale and ACM_Adapt.
> # The latter is a list of angular momentum values, and the former
> # is a list, each item of which pertains to the corresponding
> # angular momentum value, and itself is a list of eigenvalues.
> # The lowest eigenvalue across all angular momenta is obtained and
> # displayed, and all values displayed are taken with respect to it.
> # Eigenvalues are displayed for each angular momentum in the
> # range L_min..L_max, with those of constant angular momentum
> # displayed as a horizontal list. to_show restricts the maximum
> # number of eigenvalues displayed for each angular momentum.
> # The relative eigenvalues are displayed after being scaled
  (divided)
> # by the value of the global parameter glb_eig_sft.
> # The value returned is the lowest eigenvalue (unscaled).
```

```
> # If L_max is omitted then a single value L_min is used. If both
> # are omitted then all values are used (well, 0..1000000).

> Show_Eigs:=proc(eigen_vals::list,Lvals::list,
>                      toshow::nonnegint:=glb_eig_num,
>                      L_min::nonnegint:=0, L_max::nonnegint:=
  1000000,$)
>       local LL,i,eigen_low,L_count,L_top;
>       global glb_low_pre,glb_rel_wid,glb_rel_pre,glb_eig_sft,
  glb_eig_rel;

>   if toshow=0 or nops(eigen_vals)=0 then return NULL fi:

>   if _npassed=4 then   # only in this case, use a single value.
>     L_top:=L_min:
>   else
>     L_top:=L_max:
>   fi:

>   L_count:=nops(Lvals):

>   # Count how many L to output in range

>   i:=0:
>   for LL to L_count do
>     if Lvals[LL]>=L_min and Lvals[LL]<=L_top then
>       i:=i+1:
>     fi:
>   od:

>   if i=0 then
>     return NULL
>   fi:


>   if glb_eig_rel then   # use relative eigenvalues

>       # Find smallest eigenvalue across all L spaces (assuming all
  are real!)
>       # Each sublist is assumed to be increasing.

>     eigen_low:=min_head(eigen_vals);

>     printf("Lowest eigenvalue is %.*f. Relative eigenvalues follow"
>              " (each divided by %.*f):\n",
>                       glb_low_pre,eigen_low,glb_low_pre,
  glb_eig_sft);
>   else
>     eigen_low:=0;

>     printf("Scaled eigenvalues follow (each divided by %.*f):\n",
>                       glb_low_pre,glb_eig_sft);

>   fi:


>   # display all required eigenvalues, with scaling given by
  glb_eig_sft.

>   for LL to L_count do
>     if Lvals[LL]>=L_min and Lvals[LL]<=L_top then
>       # print L and first relative eigenvalue
```

```
>       printf("  At L=%2d: [%*.*f",Lvals[LL],glb_rel_wid,
  glb_rel_pre,
>                         (eigen_vals[LL][1]-eigen_low)/glb_eig_sft);
>       # print remaining eigenvalues
>       for i from 2 to min(nops(eigen_vals[LL]),toshow) do
>         printf(",%*.*f",glb_rel_wid,glb_rel_pre,
>                         (eigen_vals[LL][i]-eigen_low)/glb_eig_sft);
>       od:
>       # finish writing line
>       printf("]\n");
>     fi:
>   od:

>   eigen_low;   # return smallest eigenvalue (in case it's needed!)
> end;


> # The following functions min_head and fsel are used to obtain,
> # within a list of lists, the minimal value amongst the first
  elements.
> # These procedures are only used within Show_Eigs above.

> min_head:=(alist)->min(op(map(fsel,alist)));
> fsel:=(nlist)->`if`(nops(nlist)>0,nlist[1],NULL);


> ##################################################################
  ########

> # The procedure Show_Mels below is a very versatile procedure for
> # displaying functions of (alternative) SO(3)-reduced matrix
  elements,
> # in a convenient format, with the actual elements displayed
> # specified in the list mel_lst of "designators". NULL is returned.
> # This procedure is called directly by the procedures Show_Rats
> # and Show_Amps, which are intended to display transition rates
> # and transition amplitudes respectively (but can do otherwise)
> # calculated from the matrix elements.
> # The argument Lvals is a list of angular momenta. The argument
> # Melements is a Matrix of which each element is itself a Matrix
> # which pertains to a particular pair of angular momenta.
> # Specifically, if ki and kf are such that Li=Lvals[ki] and Lf=
  Lvals[kf]
> # then Melements[kf,ki][nf,ni] is assumed to be the alternative
> # SO(3)-reduced matrix element
> #
> #           <nf,Lf || W || ni,Li>
> #           --------------------
> #               sqrt (2*Lf+1)
> #
> # If we denote such a value by Mele, then the value actually
> # displayed is mel_fun(Li,Lf,Mele)/scale
> # (the argument mel_fun should itself be a procedure taking three
  arguments).

> # The values of Li, Lf, ni and nf for which values are displayed
> # is determined by the list showlist, each element of which should
> # itself be a list of up to five integers:
> #   1. A quartet [Li,Lf,ni,nf] designates the output of a single
  value;
> #   2. A triple of the form [Li,Lf,nf] outputs a vector of values
> #      formed from all possible values of ni (note strange order).
```

```
> #   3. A pair of the form [Li,Lf].
> #      Then a sequence of vectors of the above form is displayed
> #      for all possible values of nf.
> #   4. A single value [Lf]. Then sequence of vectors for
> #      [Li,Lf,nf] is output for all possible Li and nf.
> #   5. An empty list []. Then sequence of vectors for
> #      [Li,Lf,nf] is output for all possible Li, Lf and nf.
> #   6. A quintet of the form [Li,Lf,ni,nf,L_mod].
> #      This produces all quartets [LiX,LfX,ni,nf]
> #      with LiX=Li+k*L_mod and LfX=Lf+k*L_mod for k>=0.
> # Note that if either Li and Lf vary, then only those values are
  used
> # that differ by at most the value of the global variable
  glb_rat_TRopAM.
> # This is because glb_rat_TRopAM is (intended to be) the angular
> # momentum of the operator which produced the matrix elements
  Melements,
> # and therefore other angular momenta would give zero matrix
  elements.

> # The argument toshow gives the maximum number of values to be
  shown
> # in the case of lists produced by the designators with 3 or fewer
  items.
> # The argument mel_format is a C-style format specification,
> # which should contain two "%s" specifiers. The first will be
> # substituted for by a string that gives the two states mapped
  between
> # (this is determined by the global variable glb_tran_format which,
> # in the default implementation, takes the form "#(#) -> #(#)"),
> # and the second by the value of the function of the matrix
  element,
> # calculated as above.
> # The argument mel_desg contains a simple phrase (such as "matrix
  elements")
> # used to introduce the output.

> # Note that the default values are determined each time the
  procedure
> # is invoked, and not when it is initially defined.

> # Also note that if the 3rd argument is not a list of lists,
> # or the 5th argument is not a procedure then an error will result
> # (with that argument taken to be the 7th and an excess of
  arguments
> # being flagged).


> Show_Mels:=proc(Melements::Matrix, Lvals::list,
>                 mel_lst::list(list),
>                 toshow::integer:=glb_rat_num,
>                 mel_fun::procedure:=def_mel_fun,
>                 scale::constant:=1.0,
>                 mel_format::string:=def_mel_format,
>                 mel_desg::string:=def_mel_desg,$)

>       local L1,L2,n1,n2,L1_off,L2_off,rate_ent,TR_matrix,
>             TR_cols,TR_rows,Lmod,Lcount,item_preformat,tran_fmat1,
  tran_fmat2:
>       global glb_low_pre,glb_rel_wid,glb_rel_pre,
>              glb_rat_TRopAM,
>              glb_tran_format,glb_tran_fill,  # for "#(#) -> #(#)"
```

```
>              glb_mel_f1,glb_mel_f2,  # These three are set here
>              glb_item_format:        #  "


>   if nops(mel_lst)=0 then return NULL fi:

>   if ColumnDimension(Melements)=0 then
>     error "No matrix elements avaiable!"
>   fi:

>   printf("Selected %s follow"
>          " (each divided by %.*f):\n", mel_desg,glb_low_pre,evalf
  (scale));

>   # Specify format for the printing of each transition
  rate/amplitude.
>   # Two stages - first sets the width and precision for values to
  output.

>   item_preformat:= "%%%d.%df":
>   glb_item_format:=sprintf(item_preformat,glb_rel_wid,glb_rel_pre):

>   # Change the %s specifications in glb_tran_fmat to either
>   # "%d" for integers, or a filler which is required for the lists.

>   tran_fmat1:=sprintf(glb_tran_format,"%d","%d","%d","%d"):
>   glb_mel_f1:=sprintf(mel_format,tran_fmat1,glb_item_format):

>   tran_fmat2:=sprintf(glb_tran_format,"%d",glb_tran_fill,"%d","%d")
  :
>   glb_mel_f2:=sprintf(mel_format,tran_fmat2,"%s"):

>   # Now output the (scaled) matrix elements designated in mel_lst.
>   # Note that those in the list that are not in the
>   # range of those calculated are silently ignored.

>   for rate_ent in mel_lst do

>     if nops(rate_ent)>5 then
>       printf("  Bad matrix element specification: %a\n",rate_ent):
>       next:
>     fi:

>     if nops(rate_ent)>1 then
>       L1:=rate_ent[1]:
>       L2:=rate_ent[2]:

>       if L1<0 or L2<0 then next fi:
>     fi:


>     if nops(rate_ent)=4         # output 4-index specifiers
>         then

>       # Locate indices in Lvals for these Ls.

>       if member(L1,Lvals,'L1_off') and member(L2,Lvals,'L2_off')
  then
>         TR_matrix:=Melements[L2_off,L1_off];
>         TR_cols:=ColumnDimension(TR_matrix);
>         TR_rows:=RowDimension(TR_matrix);
```

```
>          n1:=rate_ent[3]:
>          n2:=rate_ent[4]:

>          if n1>0 and n2>0 and n1<=TR_cols and n2<=TR_rows then
>              printf(glb_mel_f1,L1,n1,L2,n2,
>                     evalf(mel_fun(L1,L2,TR_matrix[n2,n1])/scale)):
>              printf("\n"):
>          fi:
>        fi:       # L1 & L2 members


>      elif nops(rate_ent)=5          # output 5-index specifiers
>           then

>        n1:=rate_ent[3]:
>        n2:=rate_ent[4]:
>        if n1<=0 or n2<=0 then next fi:
>        Lmod:=rate_ent[5]:

>        if Lmod>0 then    # Put Lcount+1 as number of rates required
>            Lcount:=iquo(Lvals[-1]-max(L1,L2),Lmod):
>        elif Lmod<0 then
>            Lmod:=-Lmod:
>            Lcount:=iquo(min(L1,L2),Lmod):
>            L1:=L1-Lcount*Lmod:
>            L2:=L2-Lcount*Lmod:
>        else
>            Lcount:=0:
>        fi:

>        while Lcount>=0 do    # loop through all Lcount+1 cases

>          if member(L1,Lvals,'L1_off') and member(L2,Lvals,'L2_off')
  then

>            TR_matrix:=Melements[L2_off,L1_off];
>            TR_cols:=ColumnDimension(TR_matrix);
>            TR_rows:=RowDimension(TR_matrix);

>            if n1<=TR_cols and n2<=TR_rows then
>                printf(glb_mel_f1,L1,n1,L2,n2,
>                       evalf(mel_fun(L1,L2,TR_matrix[n2,n1])/scale)):
>                printf("\n"):
>            fi:
>          fi:

>          L1:=L1+Lmod:
>          L2:=L2+Lmod:
>          Lcount:=Lcount-1;
>        od


>      elif nops(rate_ent)=3    # output 3-index specifiers
>          then

>        Show_Mels_Row(Melements,Lvals,L1,L2,rate_ent[3],toshow,
  mel_fun,scale):
>                                                    # 5th arg ->
  n2

>      elif nops(rate_ent)=2    # output 2-index specifiers
>          then
```

```
>       for n2 from 1 to toshow while
>         Show_Mels_Row(Melements,Lvals,L1,L2,n2,toshow,mel_fun,
  scale)>0 do
>       od:  # keep increasing n2 until no output

>     elif nops(rate_ent)=1    # output 1-index specifiers
>         then

>       L2:=rate_ent[1]:   # note switch
>       if L2<0 then next fi:

>       # now loop through all possible L1 for |L1-L2] in TR range.

>       for L1 from max(0,L2-glb_rat_TRopAM) to L2+glb_rat_TRopAM do
>       for n2 from 1 to toshow while
>         Show_Mels_Row(Melements,Lvals,L1,L2,n2,toshow,mel_fun,
  scale)>0 do
>       od:  # keep increasing n2 until no output
>       od:

>     elif nops(rate_ent)=0   # output 0-index specifiers
>         then

>       for L2 in Lvals do

>       # now loop through all possible L1 for |L1-L2] in TR range.

>       for L1 from max(0,L2-glb_rat_TRopAM) to L2+glb_rat_TRopAM do
>       for n2 from 1 to toshow while
>         Show_Mels_Row(Melements,Lvals,L1,L2,n2,toshow,mel_fun,
  scale)>0 do
>       od:  # keep increasing n2 until no output
>       od:
>       od:

>     fi:
>   od:
>   NULL;
> end;


> # The following procedure Show_Mels_Row is used by the above
  procedure
> # Show_Mels to display, for the fixed values L1,L2,n2, the
  functions
> # of the matrix elements calculated for [L1,L2,n1,n2].
> # These are displayed as a horizontal list.
> # The arguments are the same as for Show_Mels.

> Show_Mels_Row:=proc(Melements::Matrix, Lvals::list,
>                     L1::nonnegint,L2::nonnegint,n2::integer,
>                     toshow::nonnegint,
>                     mel_fun::procedure,
>                     scale::constant,$)
>       local n1,L1_off,L2_off,col_count,
>             TR_matrix,TR_cols,TR_rows:
>       global glb_mel_f2,glb_item_format: # These two are set in
  Show_Mels.


>   if not member(L1,Lvals,'L1_off') or not member(L2,Lvals,'L2_off')
  then
```

```
>     return 0:
>   fi:

>   TR_matrix:=Melements[L2_off,L1_off];
>   TR_cols:=ColumnDimension(TR_matrix);
>   TR_rows:=RowDimension(TR_matrix);

>   if n2>TR_rows or TR_cols=0 then
>     return 0:
>   fi:

>   col_count:=min(TR_cols,toshow):

>   printf(glb_mel_f2,L1,L2,n2,
>            cat("[", sprintf(glb_item_format,
>                  evalf(mel_fun(L1,L2,TR_matrix[n2,1])/scale)),
>            seq(cat(",", sprintf(glb_item_format,
>                  evalf(mel_fun(L1,L2,TR_matrix[n2,n1])/scale))),
>                                        n1=2..col_count),"]")):

>   printf("\n"):

>   return 1:

> end:


> # The procedures Show_Rats and Show_Amps below call Show_Mels
> # above with its final four arguments (of eight) taking
> # particular values specified by certain global variables.
> # Thus the description of Show_Mels applies here.
> # For Show_Rats, the alternative SO(3)-reduced matrix element
> #
> #           <nf,Lf || W || ni,Li>
> #           --------------------
> #               sqrt (2*Lf+1)
> #
> # is displayed after being acted on by the function given by
> # the procedure glb_rat_fun, and then divided by the scale factor
> # glb_rat_sft. Each value output is displayed using the format
> # given in glb_rat_format; and the phrase given in glb_rat_desg
> # is used to introduce the output.
> # The procedure Show_Amps is similar, except using different
> # global values: the procedure glb_amp_fun is the function,
> # glb_amp_sft is the scaling factor, glb_amp_format is the format,
> # and glb_amp_desg is the phrase.

> # Both Show_Rats and Show_Amps take the required arguments
  Melements
> # and Lvals, which are as for Show_Mels. Their third arguments are
> # lists of format designators: if not given, the global variables
> # glb_rat_lst and glb_amp_lst respectively are used instead.
> # Their fourth arguments specify the maximum number of values to
> # display in a list: if not given, the global variables glb_rat_num
> # and glb_rat_num respectively are used instead.
> # Both these procedures return NULL.

> Show_Rats:=proc(Melements::Matrix, Lvals::list,
>                  rat_lst::list(list):=glb_rat_lst,
>                  toshow::integer:=glb_rat_num,$)

>       global glb_rat_sft, glb_rat_fun, glb_rat_format,
```

```
  glb_rat_desg;

>   Show_Mels(Melements,Lvals,
>                 rat_lst,
>                 toshow,
>                 glb_rat_fun,
>                 glb_rat_sft,
>                 glb_rat_format,
>                 glb_rat_desg):
> end:

> Show_Amps:=proc(Melements::Matrix, Lvals::list,
>                   amp_lst::list(list):=glb_amp_lst,
>                   toshow::integer:=glb_amp_num,$)

>       global glb_amp_sft, glb_amp_fun, glb_amp_format,
  glb_amp_desg;

>   Show_Mels(Melements,Lvals,
>                 amp_lst,
>                 toshow,
>                 glb_amp_fun,
>                 glb_amp_sft,
>                 glb_amp_format,
>                 glb_amp_desg):
> end:


> ##################################################################
  ########

> # The following procedure ACM_ScaleOrAdapt combines many of those
> # previously described to provide a versatile user-friendly
> # means of analysing Hamlitonians, displaying their eigenvalues,
> # and calculating and displaying transition rates and amplitudes
> # of the operator in the global variable glb_rat_TRop (which is
> # the quadrapole operator in the default implementation).
> # This procedure is conveniently used through the procedures
  ACM_Scale
> # and ACM_Adapt, given below, which simply set the arguments
  fit_eig
> # and fit_rat, and thus work in the same way.
> # Much of the functionality is controlled by values of the
> # global parameters.

> # The Hamiltonian is specified in ham_op; the truncated Hilbert
  space
> # is specified by the arguments anorm, lambda_base, nu_min, nu_max,
> # v_min, v_max, L_min, L_max as above.
> # The final argument L_max is optional - if omitted then L_max=
  L_min,
> # so that a single angular momentum value is used.
> # The values of fit_eig and fit_rat determine how the values that
  are
> # displayed are scaled.
> # If the argument fit_eig is zero then the relative eigenvalues
> # (relative to the lowest value) are divided by the current values
  of
> # the global parameter glb_eig_sft. If fit_eig is non-zero then the
> # value of glb_eig_sft is first determined so that the relative
  eigenvalue
> # of the (glb_eig_idx)th state of angular momentum glb_eig_L comes
```

```
  out
> # to be glb_eig_fit.
> # If the argument fit_rat is zero then the transition rates are
  divided
> # by the current values of the global parameter glb_rat_sft, and
  the
> # transition amplitudes are divided by the global parameter
  glb_amp_sft.
> # If fit_rat is non-zero then the value of glb_rat_sft is first
  determined
> # so that the transition rate from the (glb_rat_1dx)th state of
> # angular momentum glb_eig_L1 to the (glb_rat_2dx)th state of
  angular
> # momentum glb_eig_L2 comes out to be glb_rat_fit. In this latter
  case,
> # the scale parameter glb_amp_sft is then determined from
  glb_rat_sft
> # using the procedure given in glb_amp_sft_fun (it is the square
  root
> # by default).

> # Eigenvalues of the Hamiltonian are displayed using the procedure
> # Show_Eigs, and is thus of the format described for that procedure
> # above. The number of eigenvalues displayed for each angular
  momentum
> # is restricted to glb_eig_num.
> # Transition rates and amplitudes are displayed using the
  procedures
> # Show_Rats and Show_Amps, and thus have the format described above
  for
> # these procedures, the values displayed being determined by
  functions
> # given in the procedures glb_rat_fun and glb_amp_fun respectively.
> # The transition rates that are displayed are determined by the
> # designations listed in the global variable glb_rat_lst.
> # When lists of values are designated, the maximum number of states
> # for each angular momentum is restricted to glb_rat_num.
> # The transition amplitudes that are displayed are determined by
  the
> # designations listed in the global variable glb_amp_lst.
> # When lists of values are designated, the maximum number of states
> # for each angular momentum is restricted to glb_amp_num.

> # The return value is the triple
> #                 [eigen_vals,Melements,Lvals],
> # where eigen_vals is a list of lists of eigenvalues (one list for
  each
> # L-space in Lvals), and Melements are the alternative SO(3)-
  reduced matrix
> # elements of transition rates of the operator glb_rat_TRop
> # (calculated in AmpXspeig) stored as a block matrix.
> # This latter is only calculated when either of the lists
  glb_rat_lst
> # (of transition rate designators) or glb_amp_lst (or transition
> # amplitude designators) is non-empty. Otherwise, Melements is set
> # to be a 0x0 Matrix, indicating that none are available.
> # The first and third elements of the return value may be used as
  the
> # first two arguments to Show_Eigs and the second and third as the
  first
> # two arguments to Show_Rats and Show_Amps to display further
  eigenenergies,
```

```
> # transition rates/amplitudes without the need for recalculation.


> ACM_ScaleOrAdapt:=proc(fit_eig::nonnegint,fit_rat::nonnegint,
>                        ham_op::list,
>                        anorm::algebraic, lambda_base::algebraic,
>                        nu_min::nonnegint, nu_max::nonnegint,
>                        v_min::nonnegint, v_max::nonnegint,
>                        L_min::nonnegint, L_max::nonnegint,$)
>       local eigen_quin,tran_mat,Lvals,eigen_low,L_mx,L1_off,L2_off;
>       global glb_eig_num, glb_rat_lst, glb_amp_lst,
>              glb_eig_fit, glb_eig_L, glb_eig_idx, glb_eig_rel,
>              glb_rat_TRop, glb_rat_fun,
>              glb_rat_num, glb_amp_num,
>              glb_rat_fit, glb_rat_L1, glb_rat_1dx, glb_rat_L2,
  glb_rat_2dx,
>              glb_amp_sft_fun,
>              glb_eig_sft, glb_rat_sft, glb_amp_sft; # these might
  be set here

>   if _npassed<11 then  # no L_max
>     L_mx:=L_min;
>   else
>     L_mx:=L_max
>   fi:

>   # When fitting values (if either fit_eig or fit_rat is non-zero)
>   # we must ensure that the eigenvalue or transition rate with
>   # respect to which we fit, and thus choose scaling parameters,
>   # will actually be obtained. If not, we exit with an error
  message.
>   # Note that we only perform this check when values of each
  variety
>   # will actually be output (glb_eig_num>0 and nops(glb_rat_lst)>0
  resp.)

>   # First check the eigenenergy parameters

>   if fit_eig>0 and glb_eig_num>0 then
>     if glb_eig_L<L_min or glb_eig_L>L_mx or
>          glb_eig_idx>dimXspace(nu_min,nu_max,v_min,v_max,glb_eig_L)
  then
>       error "Reference state %1(%2) not available", glb_eig_L,
  glb_eig_idx;
>     fi:
>   fi:

>   # Now check the parameters for the transition rates

>   if fit_rat>0 and nops(glb_rat_lst)>0 then
>     if glb_rat_L1<L_min or glb_rat_L1>L_mx or
>          glb_rat_1dx>dimXspace(nu_min,nu_max,v_min,v_max,
  glb_rat_L1) then
>       error "Reference state %1(%2) not available", glb_rat_L1,
  glb_rat_1dx;
>     fi:

>     if glb_rat_L2<L_min or glb_rat_L2>L_mx or
>          glb_rat_2dx>dimXspace(nu_min,nu_max,v_min,v_max,
  glb_rat_L2) then
>       error "Reference state %1(%2) not available", glb_rat_L2,
  glb_rat_2dx;
```

```
>     fi:
>   fi:

>   # diagonalise the Hamiltonian on the specified space.
>   # output is [ eigenval_list, Lvals, Ps, Xparams ].

>   eigen_quin:=DigXspace(ham_op,_passed[4..-1]):
>   Lvals:=eigen_quin[4]:

>   if glb_eig_num>0 then    # require eigenvalue output

>     if fit_eig>0 then   # determine global scale factor for
  eigenvalues

>       if glb_eig_rel then  # take eigenvalues relative to smallest
>         eigen_low:=min_head(eigen_quin[1]);   # smallest eigenvalue
>       else               # take relative to 0
>         eigen_low:=0;
>       fi:

>       member(glb_eig_L,Lvals,'LL'):   # find index for required L
>       glb_eig_sft:=(eigen_quin[1][LL][glb_eig_idx]-eigen_low)
  /glb_eig_fit:

>       if glb_eig_sft=0 then
>         error "Cannot scale: reference state %1(%2) has lowest
  energy",
>                                                 glb_eig_L,
  glb_eig_idx;
>       fi
>     fi:

>     # display all required eigenvalues, with scaling given by
  glb_eig_sft.

>     Show_Eigs(eigen_quin[1],Lvals,glb_eig_num):
>   fi:

>   # Now turn attention to the transition rates, if any are
  required...

>   if nops(glb_rat_lst)>0 or nops(glb_amp_lst)>0 then

>     # obtain raw transition amplitudes

>     tran_mat:=AmpXspeig(glb_rat_TRop,op(2..-1,eigen_quin)):

>     if fit_rat>0 then   # determine global scale factor for
  transition rates
>       member(glb_rat_L1,Lvals,'L1_off'):   # find indices for
  required Ls
>       member(glb_rat_L2,Lvals,'L2_off'):

>       glb_rat_sft:=abs(glb_rat_fun(glb_rat_L1,glb_rat_L2,
>              tran_mat[L2_off,L1_off][glb_rat_2dx,glb_rat_1dx]))
  /glb_rat_fit;

>       if glb_rat_sft=0 then
>         error "Cannot scale zero transition rate B(E2: %1(%2) -> %3
  (%4))",
>                   glb_rat_L1,glb_rat_1dx,glb_rat_L2,glb_rat_2dx,
  glb_rat_fit;
```

```
>       fi:

>       # and set scaling factor for amplitudes

>       glb_amp_sft:=glb_amp_sft_fun(glb_rat_sft);
>     fi:

>     # display required transition rates with scaling factor given
>     # by glb_rat_sft, and then that for amplitudes with scaling
  factor
>     # given by glb_amp_sft (these are global variables).

>     Show_Rats(tran_mat, Lvals, glb_rat_lst, glb_rat_num):
>     Show_Amps(tran_mat, Lvals, glb_amp_lst, glb_amp_num):

>     # return the raw data in case more are required.

>   else # set tran_mat to a NULL matrix to indicate that there are
  no
>        # matrix elements available (cannot simply set to NULL).

>     tran_mat:=Matrix(0,0):

>   fi:

>   [eigen_quin[1],tran_mat,Lvals]:

> end;


> # The following procedure ACM_Scale invokes the procedure
  ACM_ScaleOrAdapt
> # above with fit_eig=0 and fit_rat=0 so that the values of the
  scaling
> # parameters glb_eig_sft, glb_rat_sft and glb_amp_sft are used
  unchanged
> # to scale the displayed values of the eigenenergies, transition
  rates
> # and amplitudes.
> # For details, see the description of ACM_ScaleOrAdapt above.

> ACM_Scale:=proc(ham_op::list,
>                 anorm::algebraic, lambda_base::algebraic,
>                 nu_min::nonnegint, nu_max::nonnegint,
>                 v_min::nonnegint, v_max::nonnegint,
>                 L_min::nonnegint, L_max::nonnegint,$)

>   ACM_ScaleOrAdapt(0,0,_passed):
> end;


> # The following procedure ACM_Adapt invokes the procedure
  ACM_ScaleOrAdapt
> # above with fit_eig=1 and fit_rat=1 so that the values of the
  scaling
> # parameters glb_eig_sft, glb_rat_sft and glb_amp_sft are
  recalculated
> # before being used to scale the displayed values of the
  eigenenergies,
> # transition rates and amplitudes.
> # For details, see the description of ACM_ScaleOrAdapt above.
```

```
> ACM_Adapt:=proc(ham_op::list,
>                 anorm::algebraic, lambda_base::algebraic,
>                 nu_min::nonnegint,nu_max::nonnegint,
>                 v_min::nonnegint,v_max::nonnegint,
>                 L_min::nonnegint,L_max::nonnegint,$)

>   ACM_ScaleOrAdapt(1,1,_passed):
> end;


> ##################################################################
  ########
> ####-------- Aiding calculations for Hamitlonians in [RWC2009] ----
  ----####
> ##################################################################
  ########

> # Here we provide procedures that may be used instead of
> # ones above to calculate for Hamiltonians considered in [RWC2009].

> # The following procedure RWC_Ham obtains the ACM encoding of
  Hamiltonians
> # of the RWC form given in (B12) (see also (89) of [RWC2009], with
> # c1=1-2*alpha and c2=alpha, and eqns. (4.230) & (4.220) of
  [RowanWood].)

> RWC_Ham:=(B,c1,c2,chi,kappa)->
>   ACM_Hamiltonian(-1/2/B,0,B*c1/2,B*c2/2,0,-chi,0,0,0,kappa);


> # The following procedure RWC_expt gives the expectation value of
  the above
> # Hamiltonian on the |(anorm,lambda0)0;0100> basis state, given by
  (B16).
> # (Note that (76) of [RWC2009] contains typos.)

> RWC_expt:=proc(B::constant,c1::constant,c2::constant,
  kappa::constant,
>                           anorm::constant,lambda0::constant,$)
>   local aa:

>   aa:=anorm^2:
>   aa*(4+9/(lambda0-1))/8/B + B*lambda0*c1/2/aa
>              + B*lambda0*(lambda0+1)*c2/2/aa^2 + kappa/3:
> end:

> # The following procedure RWC_expt_link gives the same expectation
  value
> # (B16) as that above, but lambda0 is assumed to depend on anorm
  through
> # the function RWC_dav (see below).

> RWC_expt_link:=proc(B::constant,c1::constant,c2::constant,
  kappa::constant,
>                           anorm::constant,$)
>   RWC_expt(_passed,evalf(RWC_dav(c1,c2,anorm))):
> end:


> # The following procedure RWC_dav calculates lambda0 from anorm
> # (and c1 and c2) using (B11) via (B15).
```

```
> RWC_dav:=proc(c1::constant,c2::constant,anorm::constant,
  v::nonnegint:=0,$)
>   lam_dav(anorm,beta_dav(c1,c2),v)
> end:

> # The following calculates lambda_v using (B7) - or using
> # B11 if the final argument is not given (it defaults to 0).

> lam_dav:=proc(a::constant,beta0::constant,v::nonnegint:=0,$)
>     1+sqrt( (v+3/2)^2 + a^4*beta0^4 )
> end:

> # The following calculates beta_0 using (B15)

> beta_dav:=proc(c1::constant,c2::constant,$)
>   if evalf(c1)>=0 then
>     0
>   else
>     sqrt(-c1/c2/2)
>   fi;
> end:

> # The following procedure RWC_alam returns values of the ACM
  parameters
> # (anorm,lambda), which are "optimal" in the cases of the RWC
  Hamiltonians.
> # This seeks the minimal value of RWC_expt, given above, by solving
> # for a turning point.
> # The fourth parameter is for seniority v, which is 0 for the
> # analysis given in [WR2015], but in the case of more general v,
> # (for L=0 (so 3\v) and no spherical dependence in the potential),
> # the 9/4 factor in the first term of (B.16) is replaced by
> # the more general (v+3/2)^2.

> RWC_alam:=proc(B::constant,c1::constant,c2::constant,v::nonnegint:=
  0,$)
>   local RWC1,RWC2,muf,aa0,vshft,A;

>   vshft:=(2*v+3)^2:  # This is 9 for the v=0 case.

>   if evalf(c1)<0 then
>           # Here lambda is a function of aa (i.e. a^2).
>           # There is always one positive solution in this case
>           # (in fact, I've never found other real solns).

>     # We use mu=2(lambda-1) where lambda is given by (B11) via
  (B15).

>     muf:=(aa) -> sqrt( vshft + (aa*c1/c2)^2 ):

>     # The following is the derivative of (B16) noting that
>     # d(mu)/d(aa)=aa*(c1/c2)^2/mu.  But multiplied by 4*aa^3*B*mu.

>     RWC2:=(aa,mu) -> (c1/c2)^2 * (-vshft*aa^5/mu^2
>                                    + aa^3*B^2*c1 + aa^2*B^2*c2*
  (mu+3))
>                         + aa^3*(2*mu+vshft)
>                         - B^2*mu*(mu+2)*(aa*c1+c2*(mu+4)):

>     aa0:=max(fsolve(RWC2(A,muf(A))=0,A)):
>     return [sqrt(aa0),1+muf(aa0)/2]:
```

```
>   else   # (Here lambda is constant)
>          # There is always 1 positive solution in this case
>          # (fsolve produces real solns.) and possibly two others
  that
>          # are negative. Use max to exclude them.

>     RWC1:=(aa) -> aa^3 - B^2*c1*aa - (2*v+7)*B^2*c2:

>     aa0:=max(fsolve(RWC1(A)=0,A)):
>     return [sqrt(aa0),2.5]:
>   fi:

> end:

> # The following procedure RWC_alam36 is a simplified algorithm
> # for obtaining "optimal" values of (anorm,lambda), obtained by
> # matching second derivatives at the turning point of the
  potential.
> # This only gives good results in certain cases.

> RWC_alam36:=proc(B::constant,c1::constant,c2::constant,$)
>   local RWC1,RWC2,muf,aa0;

>   if evalf(c1)<0 then

>     return evalf([sqrt(sqrt(-B^2*c1/2)),1+sqrt(36+B^2*c1^4/c2^2)/4]
  ):

>   else

>     return evalf([sqrt(sqrt(B*c1/4)),2.5]):

>   fi:

> end:

> # The following procedure RWC_alam_clam is another alternative that
> # returns values of the ACM parameters (anorm,lambda), which are
> # obtained from the minimal value of the expectation value of
  RWC_expt,
> # given above, with lambda assumed to take the constant value of
  2.5

> RWC_alam_clam:=proc(B::constant,c1::constant,c2::constant,$)
>   local RWC1,RWC2,muf,aa0,A;

>     RWC1:=(aa) -> aa^3 - B^2*c1*aa - 7*B^2*c2:
>     aa0:=max(fsolve(RWC1(A)=0,A)):
>     return [sqrt(aa0),2.5]:

> end:


> # The following procedure RWC_alam_fun returns a triple
> #                 [anorm,lambda0,lambda_fun]
> # where anorm and lambda0 are "optimal" values obtained as in
> # RWC_alam above, and lambda_fun is a procedure that takes an
  argument
> # v that gives the (optimal) value of lambda(v)-lambda(0),
> # an integer of same parity as v.
> # This is not used elsewhere.
```

```
> RWC_alam_fun:=proc(B::constant,c1::constant,c2::constant,$)
>   local RWC1,RWC2,muf,aa0,A;

>   if evalf(c1)<0 then
>           # Here lambda is a function of aa (i.e. a^2).
>           # There is always one positive solution in this case
>           # (in fact, I've never found other real solns).

>     # We use mu=2(lambda-1) where lambda is given by (B11) via
  (B15).

>     muf:=(aa) -> sqrt( 9+ (aa*c1/c2)^2 ):

>     # The following is the derivative of (B16) noting that
>     # d(mu)/d(aa)=aa*(c1/c2)^2/mu.  But multiplied by 4*A^3*B*mu.

>     RWC2:=(aa,mu) -> (c1/c2)^2 * (-9*aa^5/mu^2
>                                      + aa^3*B^2*c1 + aa^2*B^2*c2*
  (mu+3))
>                         + aa^3*(2*mu+9)
>                         - B^2*mu*(mu+2)*(aa*c1+c2*(mu+4)):

>     aa0:=max(fsolve(RWC2(A,muf(A))=0,A)):
>     return [sqrt(aa0),1+muf(aa0)/2,lambda_davi_fun((aa0*c1/c2/2)^2)
  ]:

>   else   # (Here lambda is constant)
>          # There is always 1 positive solution in this case
>          # (fsolve produces real solns.) and possibly two others
  that
>          # are negative. Use max to exclude them.

>     RWC1:=(aa) -> aa^3 - B^2*c1*aa - 7*B^2*c2:

>     aa0:=max(fsolve(RWC1(A)=0,A)):
>     return [sqrt(aa0),2.5,lambda_sho_fun]:
>   fi:

> end:


> ##############################################################
  ###

> # We now set default values for all the global parameters

> ACM_set_defaults(0):

> ##############################################################
  ###
> ##############################################################
  ###


> ##############################################################
  ###
> ### Changes from version 1.3 to this version 1.4.
> ##############################################################
  ###

> #     1. New procedure MF_Radial_id_poly() to calculate the
  polynomial
```

```
> #        (34). Altered the procedures ME_Radial_id_pl() and
> #        ME_Radial_id_ml() to call it, and evaluate the return
  value.
> #     2. Altered the procedure load_CG_table() so that it
> #        now deals correctly with the v2=0 case: for this
> #        case only, it doesn't load data from a file but generates
> #        the coefficients: the non-zero ones all being 1.0.
> #     3. Made the variable 'A' local in procedures RWC_alam() etc.,
> #        in case it has been assigned non-locally
> #        (for then an error would occur).
> #     4. Changed the float[8] datatypes in the Matrix
> #        constructors to float so that things still work if
> #        Digits is increased beyond that for which hardware
> #        mathematical operations are possible (usually 15).
> #     5. Altered some procedure argument types numeric -> constant
> #        to make them more versatile. Then, to cope with surd
  arguments,
> #        the 'if c1<0' tests have been changed to 'if evalf(c1)<0'.
> #     6. Coding of RWC_dav has changed to call (new)
> #        procedures lam_dav and beta_dev, making extensions
> #        and testing easier. The functionality hasn't changed.
> #        Similarly, also added v argument to RWC_alam.
> #     7. Changed coding of RepRadial_Prod and RepRadial_Prod_rem
> #        to ensure that when the operator list contains SU(1,1)
> #        operators (Sm,Sp or S0), and a lambda-changing identity
> #        operator is required, it is multiplied on the left or
  right
> #        as appropriate (though such a calculation is probably
> #        meaningless, c.f. eqns. (19) & (20) ).
> #     8. The procedures RepRadial_Prod_rem, RepRadial_Prod_rem,
> #        and RepRadial_bS_DS now return errors if either of their
> #        lambda values are non-positive.
> #     9. Applied simplify to return values of RepRadial and
> #        RepRadial_param because Maple doesn't automatically
> #        perform GAMMA cancellations for large values (>100) of
> #        its arguments (one wonders why not just for large values!)
  .
> #    10. Removed a couple of unncessary Matrix assigments,
> #        because the calculations were carried out 'inplace'.

> ##########################################################################
  ###
> ###  End of Maple code file acm1.4.mpl .
> ##########################################################################
  ###
```